目錄

- Google Deep Learning Notes 1.1
- Lesson 1 Machine Learning to Deep Learning 1.2
  - Logistic Classification 1.2.1
  - Logistic Classification实践 1.2.2
  - Stochastic Optimization 1.2.3
- Lesson 2 Deep Neural Network 1.3
  - Limit of Linear Model 1.3.1
  - Neural network 1.3.2
  - 神经网络实践 1.3.3
  - Deep Network 1.3.4
  - 深度神经网络实践 1.3.5
  - 神经网络做数据分类 1.3.6
- Lesson 3 Convolutional Networks 1.4
  - 卷积神经网络实践 1.4.1
- Lessson 4 Deep Models for Text and Sequence 1.5
  - 循环神经网络实践 1.5.1
  - 理解LSTM 网络 1.5.2
- TensorFlow笔记 1.6
  - TensorFlow安装 1.6.1
  - 初识Tensorboard 1.6.2
  - SKflow 1.6.3
  - 分布式TensorFlow 1.6.4

Google Deep Learning Notes



前排提醒：工具归工具，研究归研究，AI的研究唯有打好基础，多看论文，多做实验，方能有所增益。

> Google 深度学习笔记
>
> Github工程地址：https://github.com/ahangchen/GDLnotes
>
> 欢迎star，有问题可以到Issue区讨论
>
> 官方教程地址
>
> 视频/字幕下载
>
> 教程已重构兼容TensorFlow 1.2

框架： TensorFlow （安装踩坑日志）

> 谷歌家的深度学习框架，有Python,C++的API

工具：Pycharm

笔记列表

- Lesson 1 Machine Learning to Deep Learning
  - Logistic Classification
  - Logistic Classification实践
  - Stochastic Optimization
- Lesson 2 Deep Neural Network
  - Limit of Linear Model
  - Neural network
  - 神经网络实践
  - 优化神经网络：Deep Network
  - 深度神经网络实践
- Lesson 3 Convolutional Networks
  - 卷积神经网络实践
- Lessson 4 Deep Models for Text and Sequence
  - Challenge

  - Model
  - Sequence
  - 循环神经网络实践

附录：

- NumPy笔记（待完善）
- matplotlib笔记（待完善）
- sklearn笔记（待完善）
- TensorFlow笔记

最近在整理西瓜书中的一些概念，有兴趣的同学可以参与

> 觉得我的文章对您有帮助的话，就给个star吧～

攒钱买四路1080ti台式机ing



Machine Learning to Deep Learning

深度学习

- 我们可以在Personal Computer上完成庞大的任务
- 深度学习是一种适应于各类问题的万能药

神经网络

- 神经网络出现于80年代，但当时计算机运行慢，数据集很小，神经网络不适用
- 现在神经网络回来了，因为能够进行GPU计算，可用使用的数据集也变大

分类

> 分类的一些讨论可以在这个项目里看到

- Machine Learning不仅是Classification！但分类是机器学习的核心。
- 学会分类也就学会了Detect和Rank
    - Detect：从复杂场景中识别某类物品
    - Rank：从各种链接中找到与某个关键词相关的一类链接

- Logistic Classification
- Logistic Classification实践
- Stochastic Optimization

> general data practices to train models
>
> 觉得得我的文章对您有帮助的话，就给个star吧～

Logistic Classification

> Github工程地址：https://github.com/ahangchen/GDLnotes
>
> 欢迎star，有问题可以到Issue区讨论
>
> 官方教程地址
>
> 视频/字幕下载

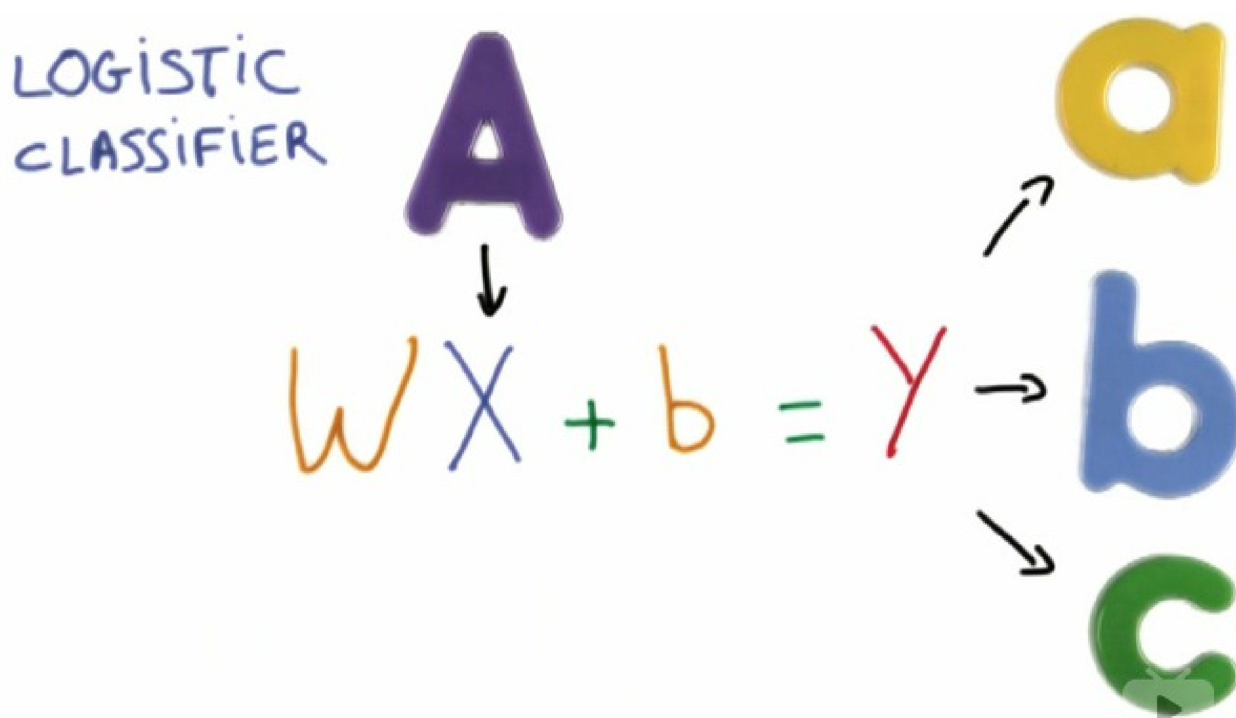
About

> simple but important classifier

- 训练你的第一个端到端模型
- 下载并预处理图片
- 在图像数据上运行Logistic classifier进行分类
- 相关的数学背景知识和代码

Detail

Linear Classifier



> 之所以这样建模，是因为线性公式是最简单的数学模型，仅此而已。

- Input: X (e.g. 图像中像素的灰度值)
- 将一个线性函数作用在X上
  - 大矩阵相乘
  - 输入一个代表图片的向量
  - 将输入向量和一个矩阵W相乘，W表示权重
  - b代表偏移（biased）项
  - 机器学习便是调整权重和偏移值以达到最好的预测效果
- 输出: Y, 对输入应当属于哪个类进行预测

  - Y是一个代表每个label可能性的向量
  - 好的预测中，正确的label的概率应当更接近1
  - 往往得到的Y一开始不是概率，而是一些具体值（scores/logits），所以需要转换，by：

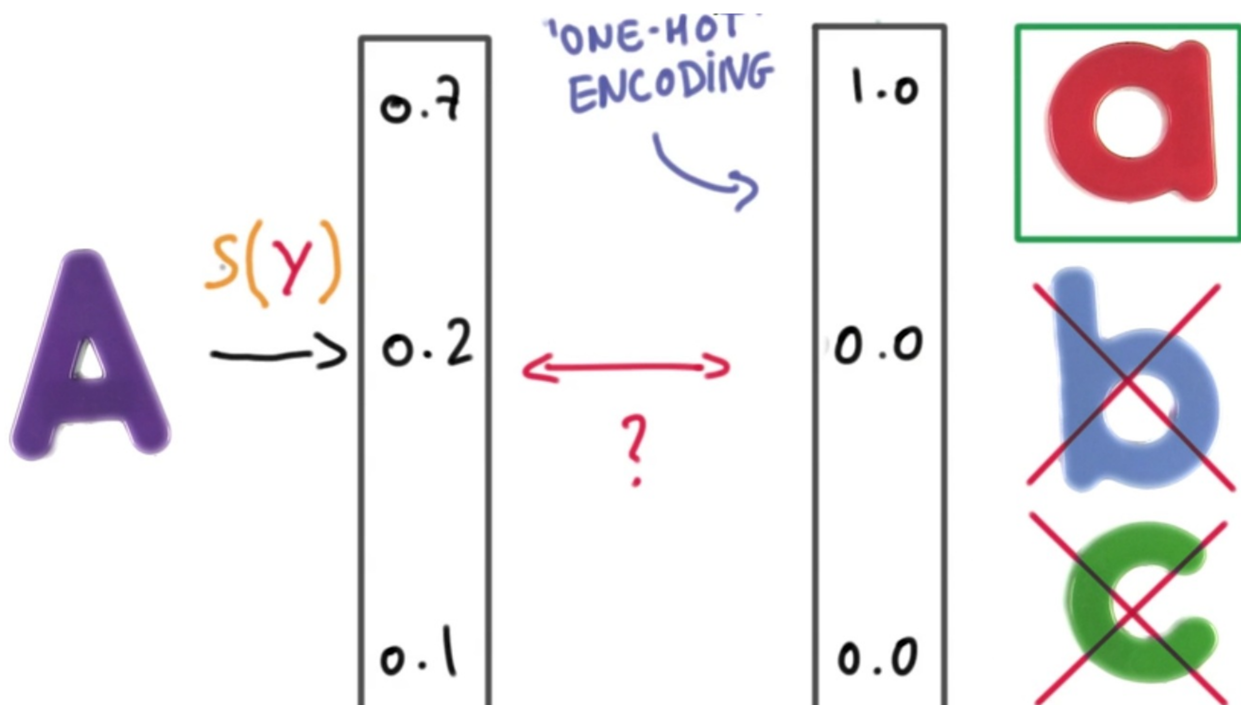
> Softmax回归模型：Wikipedia

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Softmax

- 代码 soft_max.py：Softmax实现与应用
- input的score差异越大（可以全部乘10试试），则输出的各项label概率差异越大，反之差异越小
- Softmax只关心几个label之间的概率，不关心具体值
- 机器学习是一个让预测成功率升高的事情，因此是一个让score之间差异增大的过程

One hot encoding



> 正确预测结果应当是只有一个label成立，其他label不成立。这种情况下，预测概率最大的则是最可能的结果。
>
> Example: take this test

- one hot encoding在label很多的情况下效果不好，因为output vector到处都是0，很稀疏，因此效率低
  - solved by embeddings
- 好处：可以measure我们与理想情况之间的距离（compare two vectors）

> 分类器输出：[0.7 0.2 0.1] \<=> 与label对应的真实情况：[1 0 0]

- Compare two vectors: cross-entropy

CROSS-ENTROPY

S(Y) 0.7 0.2 0.1

L 1.0 0.0 0.0

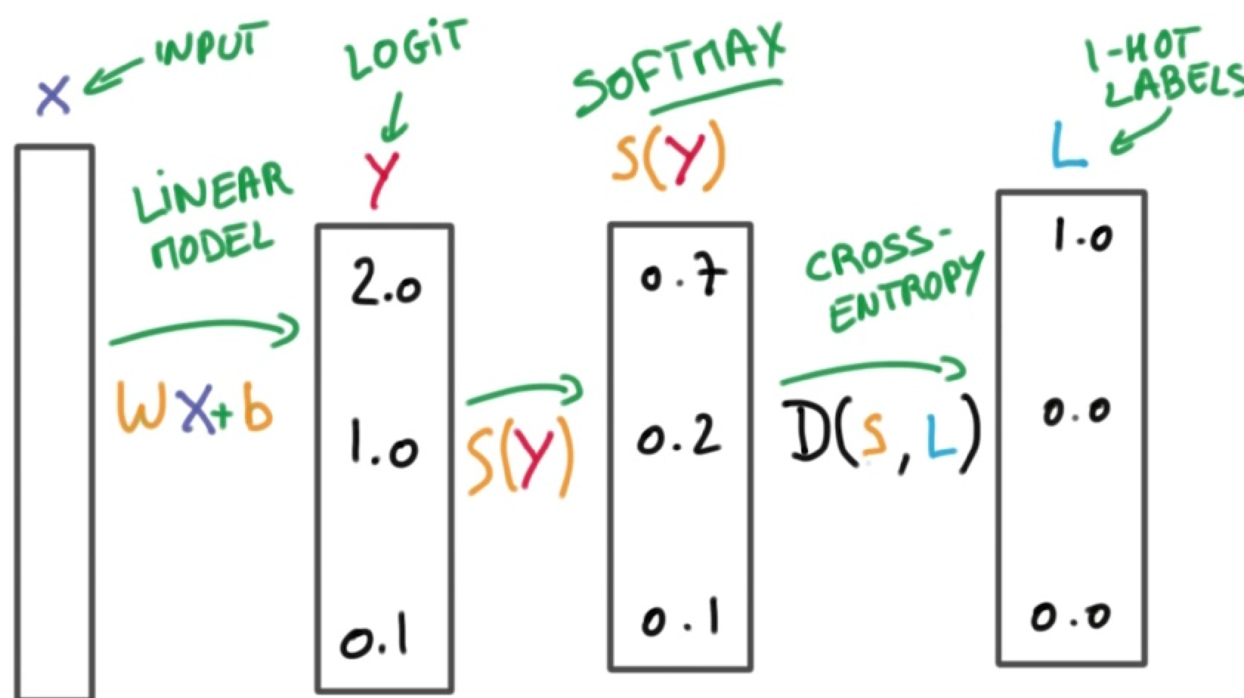
$$D(S,L) = -\sum_i L_i \log(S_i)$$

called the cross-entropy.

- D(S, L) != D(L, S)

> Remember: Label don't log, for label zero

小结



MULTINOMIAL LOGISTIC CLASSIFICATION

$$D(S(WX+b), L)$$

找到合适的W和b，使得S和L的距离D的平均值，在整个数据集n中最小。

最小化cross-entropy

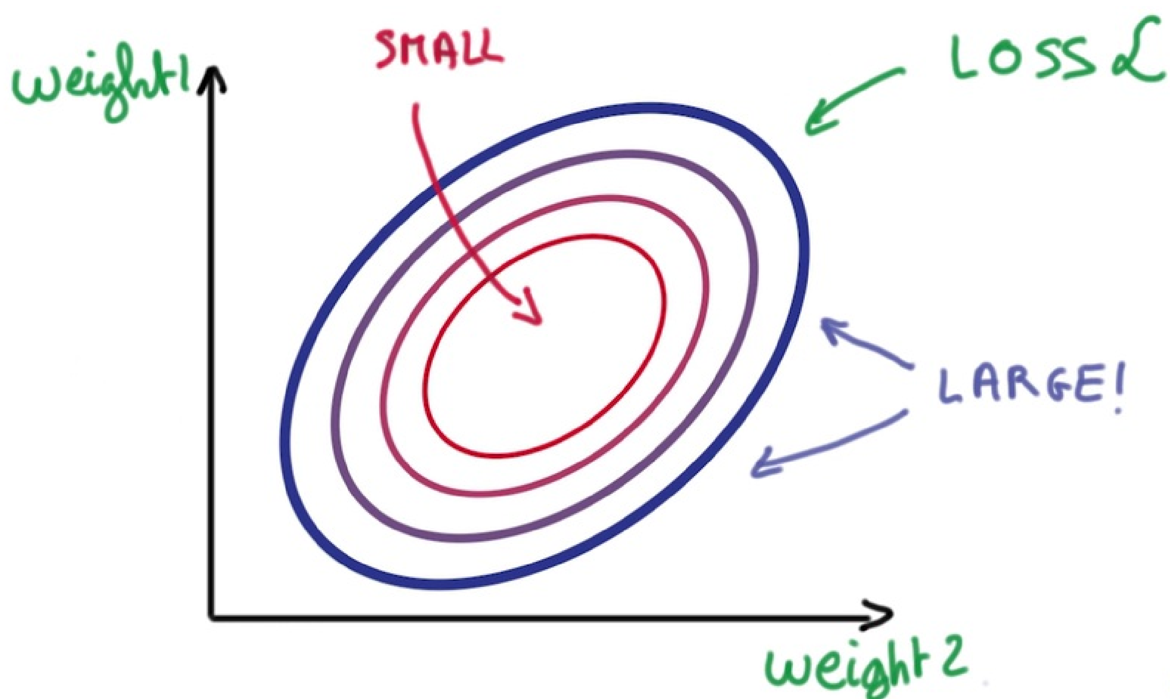
LOSS = AVERAGE CROSS-ENTROPY

BIG MATRIX!!

$$\mathcal{L} = \frac{1}{N} \sum_i D(S(wx_i + b), L_i)$$

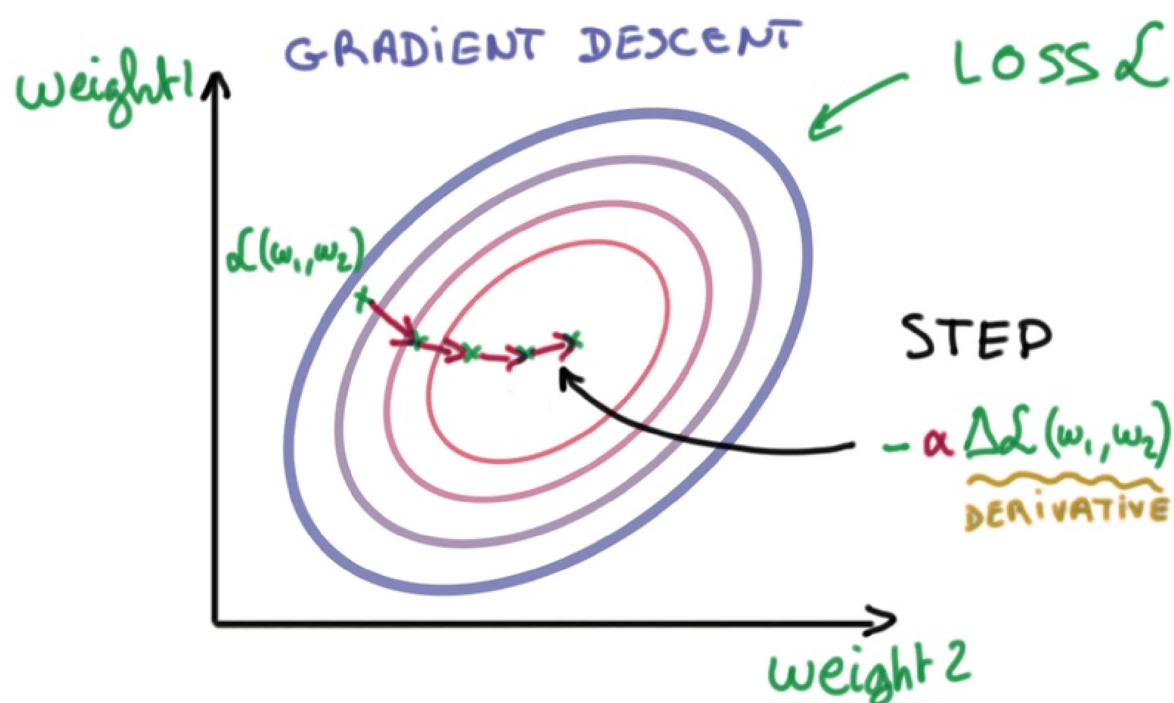
BIG SUM!!

D的平均值即是Training loss，求和和矩阵相乘是个大数据的活。



两个参数的误差导致一个呈圆形的loss，所以我们要做的就是找到尽量靠近圆心的weight

> 机器学习问题变成了一个数值优化
>
> - 解决方法之一：Gradient descent，求导



> 修改参数，检查误差是否变大，往变小的方向修改，直到抵达bottom。
>
> 图中weight是二维的，但事实上可能有极多的weight

Numerical Stability

> 量级相差太多的数运算会导致许多错误

Example:num_stable.py

- 你可能以为输出是1， 但结果是一个接近0.95的数。
- 但将1billion换成1，结果就很接近1。
- 因此需要让前面提到的Train loss函数中的数据不要too big or too small

Normalized Inputs and Initial Wights

> 归一化输入和初始参数

- 理想目标
  - 均值为0
  - 方差处处相等

MEAN

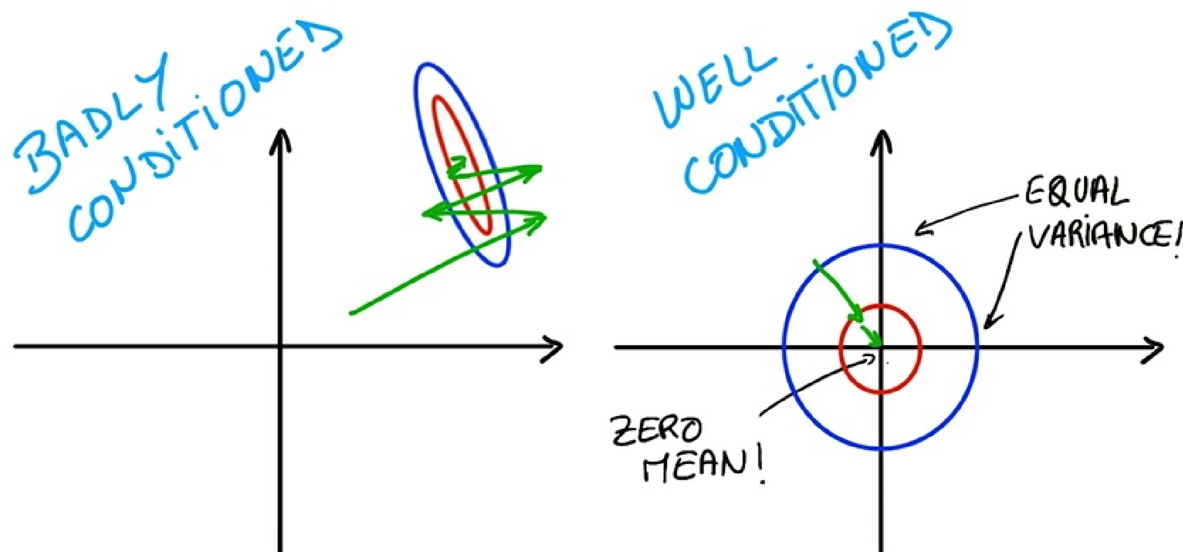
$$X_i = 0$$

VARIANCE

$$\sigma(X_i) = \sigma(X_j)$$

- Math Reason

> Easier for the optimizer to find a good solution



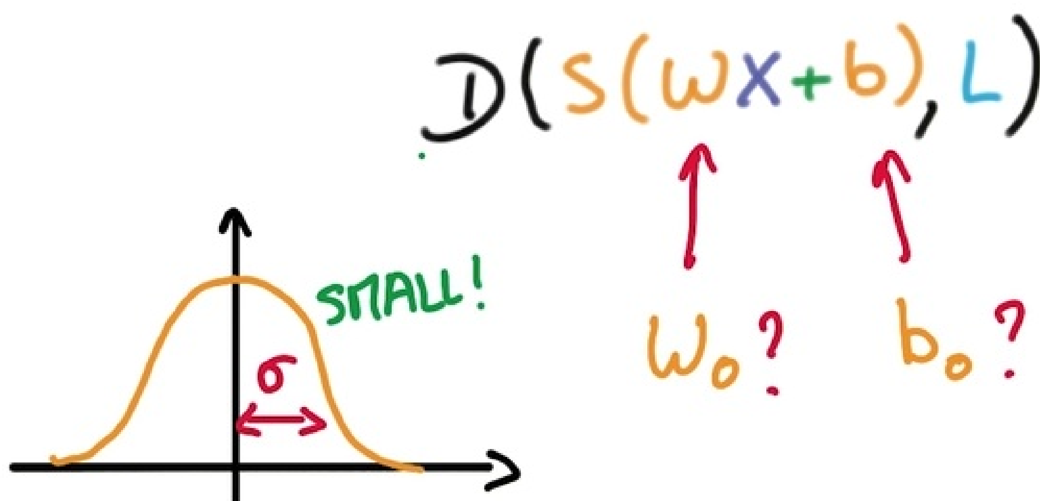
- Example: Images Normalization

```
R = (R - 128) / 128
G = (G - 128) / 128
B = (B - 128) / 128
```

- Weight Initialization 找到好的weight和bias for the gradient descent to proceed

> A simple, general method

WEIGHT INITIALIZATION



- 用均值为0，标准偏差的高斯分布产生随机的数据填充W矩阵

INITIALIZATION OF THE LOGISTIC CLASSIFIER

$$\frac{\text{pixels} - 128}{128}$$

$$\mathcal{L} = \frac{1}{N} \sum_i D(S(WX_i + b), L_i)$$

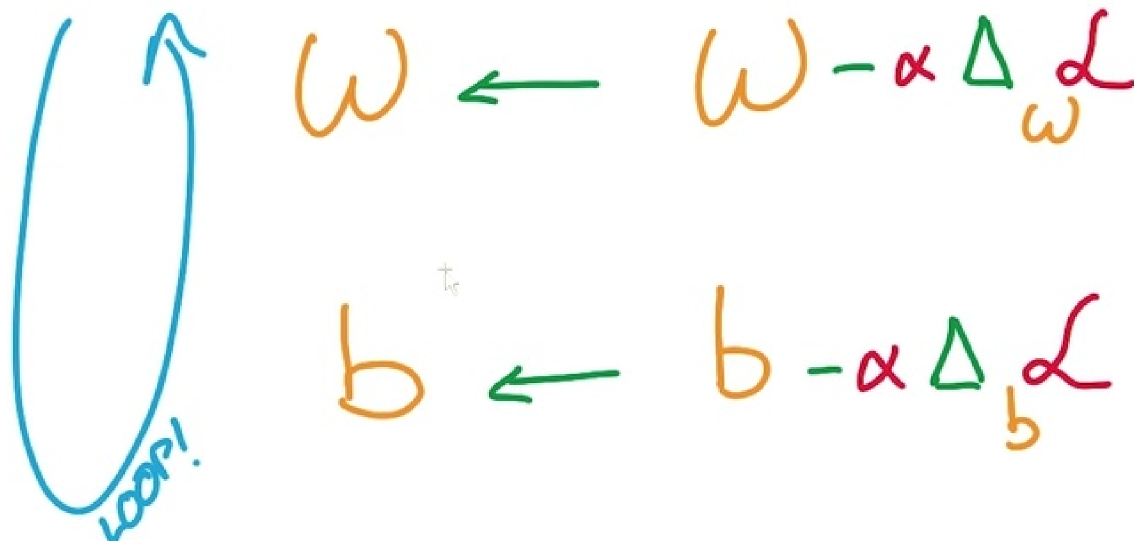
σ

ϕ

- 高斯分布模型也决定了初始输出(softmax输出)的概率分布

- 高斯分布的sigma越小，说明预测越不确定，sigma的取值很主观
- 我们的工作即是，选一个较小的sigma，让sigma变小到合适的值，使得预测更确定。

- 优化



调整W和b，使得Train loss最小

扩展阅读：西瓜书第三章·线性模型 下一节实践

> 觉得得我的文章对您有帮助的话，就给个star吧～

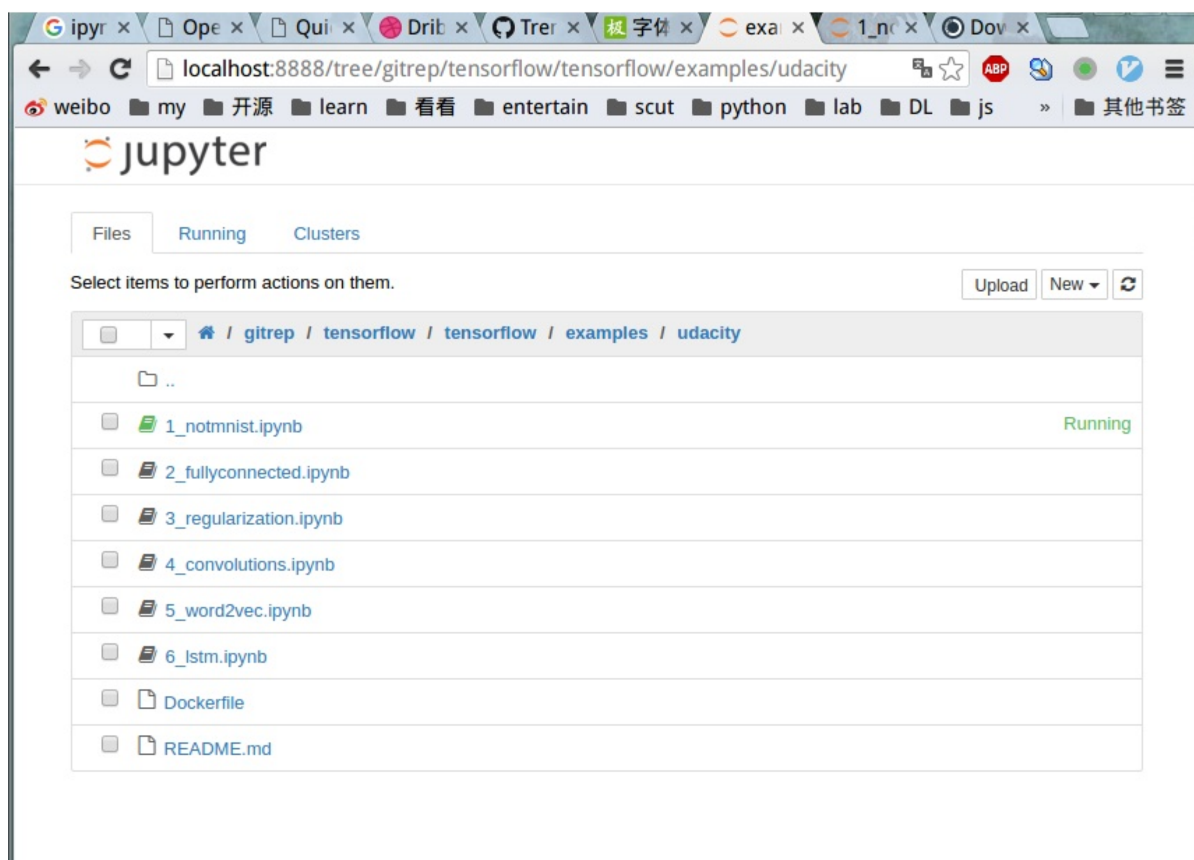
Practical Aspects of Learning

> Github工程地址：https://github.com/ahangchen/GDLnotes
>
> 欢迎star，有问题可以到Issue区讨论
>
> 官方教程地址
>
> 视频/字幕下载
>
> 课程目标：学习简单的数据展示，熟悉以后要使用的数据

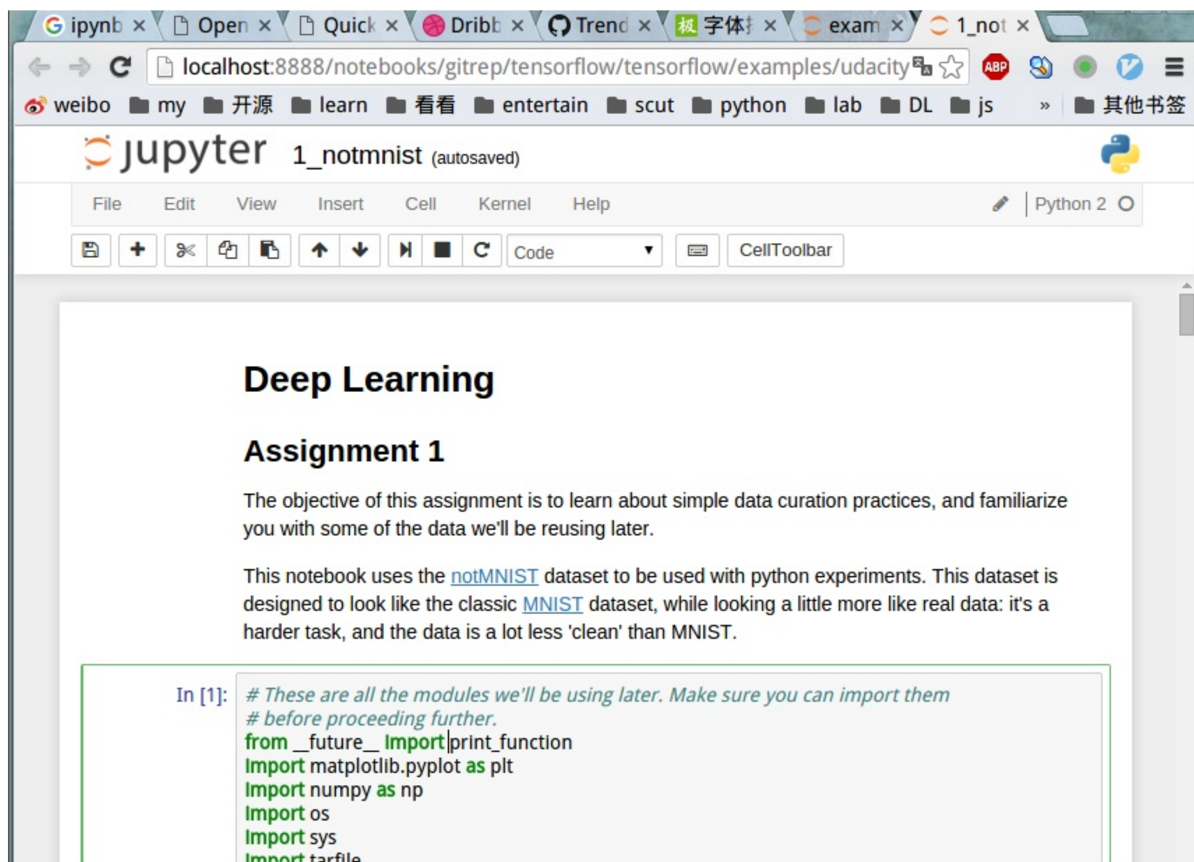
Install Ipython NoteBook

可以参考这个教程

- 可以直接安装anaconda，里面包含了各种库，也包含了ipython；
- 推荐使用python2的版本，因为很多lib只支持python2，而且python3在升级中，支持3.4还是3.5是个很纠结的问题。
- 安装anaconda后直接在终端输入 ipython notebook，则会运行一个ipython的server端，同时在你的浏览器中打开基于你终端目录的一个页面：



- 点开ipynb文件即可进入文件编辑页面



上图即为practical部分的教程，可以在github下载

官方推荐使用docker来进行这部分教程，但简单起见我们先用ipython notebook

安装tensorflow

notMNIST

修改的MNIST，不够干净，更接近真实数据，比MNIST任务更困难。

Todo

我将官方教程的一个文件拆成了多个（以文件持久化为边界），然后在schedule.py里统一调用，在各个文件里可以执行各个部分的功能测试。

- 下载
    - 使用urlretrieve来获取数据集notMNIST_large.tar.gz和notMNIST_small.tar.gz

> 代码示例：load_data.py

- 解压
    - 使用tarfile模块来解压刚刚下载的压缩包

  > 代码示例：extract.py

- 读图 - 展示 - 序列化
    - 用ndimage读取一部分图片，用pickle将读取到的对象（ndarray对象的list）序列化存储到磁盘
    - 用matplotlib.plot.imshow实现图片显示，可以展示任意的numpy.ndarray，详见show_imgs(dataset)
    - 这里展示的是二值化图片，可以设置显示为灰度图
    - 将每个class对应的图像数据集序列化到磁盘

  > 代码示例：img_pickle.py

- 整理数据集
    - 用pickle读取pickle文件，
    - 从train_folder中为10个class分别获取10000个valid_dataset和20000个train_dataset，
    - 其中对每个class读取到的数据，用random.shuffle将数据乱序化
    - 将各个class及其对应的label序列化到磁盘，分别为训练器和校验集
    - 从test_folder中为10个class分别获取10000个test_dataset,
    - 其中对每个class读取到的数据，用random.shuffle将数据乱序化
    - 将各个class及其对应的label序列化到磁盘，作为测试集

  > 代码示例merge_prune.py

- 去除重复数据
    - load_pickle，加载dataset
    - 先将valid_dataset中与test_dataset重复部分剔除，再将train_dataset中与valid_dataset重复部分剔除
    - 每个dataset都是一个二维浮点数组的list，也可以理解为三维浮点数组，
    - 比较list中的每个图，也就是将list1中每个二维浮点数组与list2中每个二维浮点数组比较
    - 示例代码即为clean_overlap.py中的imgs_idx_except

    - 我们在拿list1中的一个元素跟list2中的一个元素比较时，总共需要比较len(list1) *len(list2)* image_size * image_size次，速度极慢

    - 实际上这是有重复的计算的，就在于，list2中的每个元素，都被遍历了len(list1)次
    - 因此有这样的一个优化，我们遍历每个图，用图中的灰度值，仿照BKDRHash，得到每个图都不同的hash值，比较hash值来比较图像
    - 示例代码即为clean_overlap.py中的imgs_idx_hash_except

- 这样每个图都只需要访问一次，计算hash的时间变为(len(list1) + len(list2)) *image_size* image_size
- 比较的次数是len(list1) * len(list2)
- 由于我们的数据中，list1和list2的长度是大数，所以节省的时间是相当可观的
- 在我的机器上，比较完valid_dataset和test_dataset需要的时间分别是25000秒（10000次比较，每次2-3秒）和60秒
- 然后再将清理后的数据序列化到磁盘即可

> 代码示例：clean_overlap.py

- 训练一个logistics 模型
    - 将train_dataset作为输入，用valid_dataset进行验证（预测成功率81.9%）
    - 为了重复利用训练后的分类器，将其序列化到磁盘

> 代码示例：logistic_train.py

- Measure Performance
    - 分类器会尝试去记住训练集
    - 遇到训练集中没有的数据时，分类器可能就没辙了
    - 所以我们应该measure的是，分类器如何产生新数据（生成能力（推导能力）越大，说明它应对新数据能力越强）
    - 仅measure分类器记忆数据集的能力并不能应对新数据（没有学到规律），所以不应该拿旧数据去measure
    - 因此measure的方式应该是拿新数据去看分类器的预测准确度（never see, can't memorize）
    - 但是在measure的过程中，我们会根据测试数据去重新调整分类器，使其对所有测试数据都生效
    - 也就是说测试数据变成了训练集的一部分，因此这部分数据我们只能作为valid_dataset，而不能用于衡量最后的performance
    - 解决方法之一即，最终进行performance measure的数据集，必须是调整分类器的过程中没有使用过的
    - 即坚持一个原则，测试数据不用于训练

> 在机器学习比赛Kaggle中，有public data，validate data，并有用于测试（选手未知）的private data，只有在训练时自己的分类器时，预先取一部分数据作为test data， 才能不会在train和valid的过程中被已有数据所蒙蔽

- Validation dataset
    - 验证集越大，验证的可信度越大

- 统计学上，调整分类器后，当30个以上预测结果的正确性发生变化的话，这种变化是可信的，值得注意的，小于30是噪音
- 因此Validation dataset通常数据要大于30000个，在准确率变化高于0.1%时，认为分类器的performance变化
- 但这样需要的数据往往偏多，所以可以尝试交叉验证（cross validation），交叉验证有个缺点是速度慢
- 验证时，使用tensor.eval(input)，相当于tf.get_default_session().run(tensor)

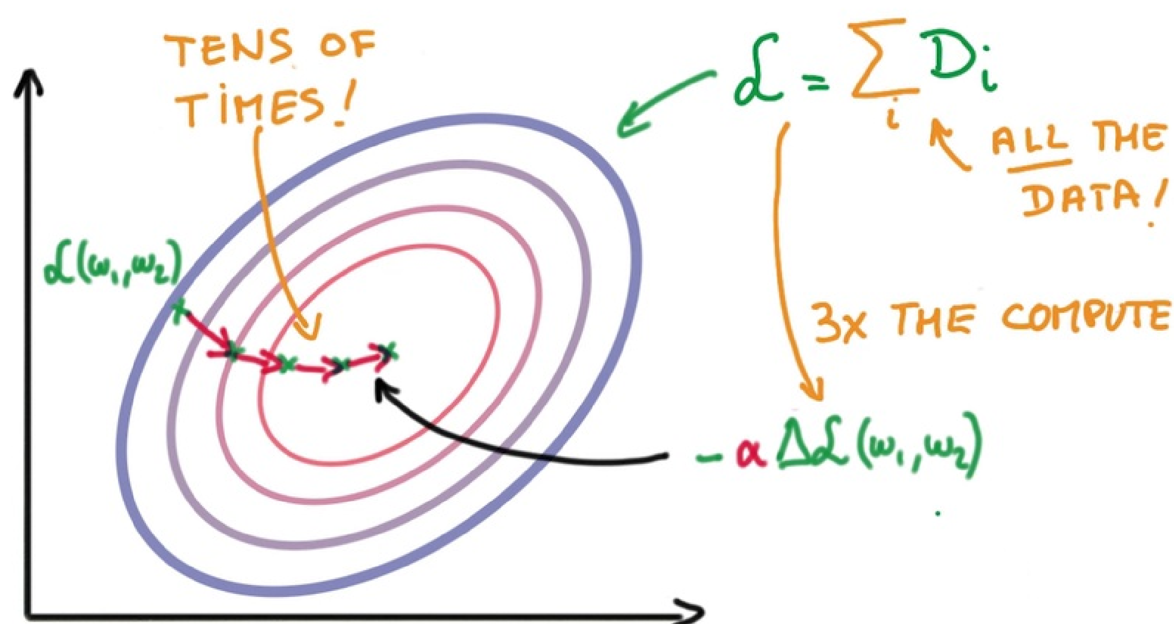
扩展阅读：西瓜书第二章·模型评估与选择

> 觉得得我的文章对您有帮助的话，就给个star吧～

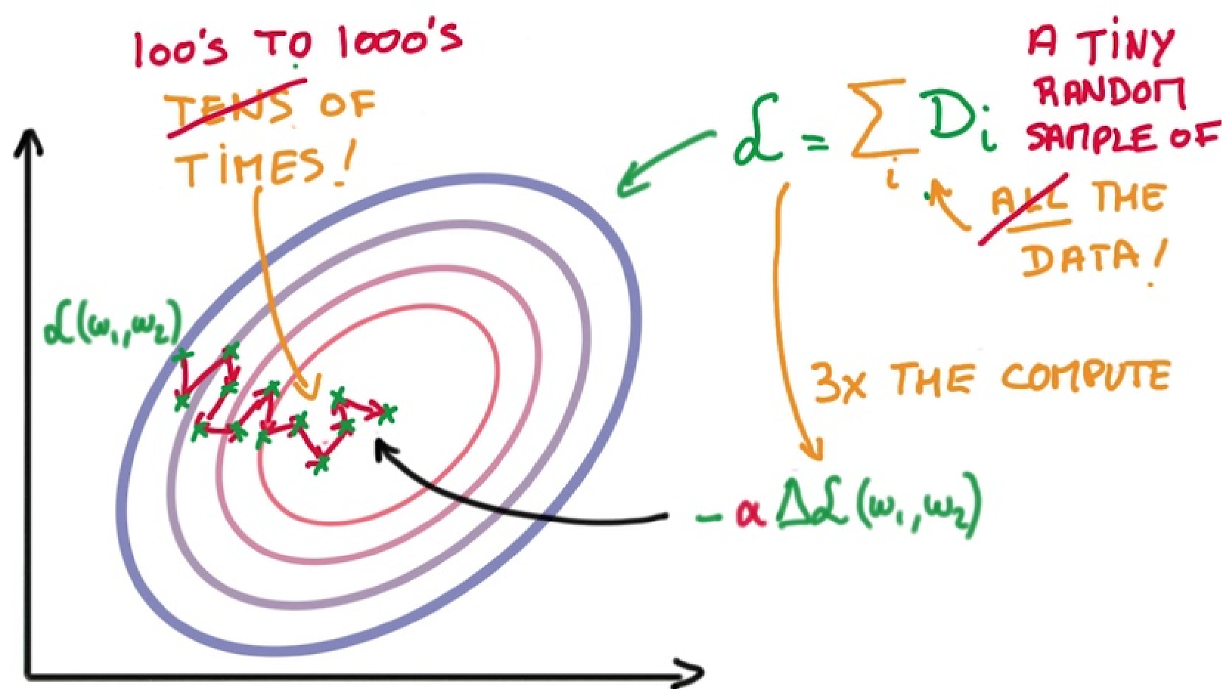
Stochastic Optimization

> Github工程地址：https://github.com/ahangchen/GDLnotes
>
> 欢迎star，有问题可以到Issue区讨论
>
> 官方教程地址
>
> 视频/字幕下载

- 实践中大量机器学习都是通过梯度算子来求优化的
- 但有一些问题，最大的问题就是，梯度很难计算
- 我们要计算train loss，这需要基于整个数据集的数据做一个计算
- 而计算使 train loss 下降最快的调整方向需要的时间是计算train loss本身的三倍



- 因此有了SGD：Stochastic Gradient Descent
  - 计算train loss时，只随机取一小部分数据集做为输入
  - 调整W和b时，调整的大小step需要比较小，因为数据集小，我们找到的不一定是对的方向
  - 这样也就增加了调整的次数
  - 但可观地减小了计算量



SGD的优化

> 实际上SGD会使得每次寻找的方向都不是很准，因此有了这些优化

- 随机的初始值

HELPING SGD

1- INPUTS

MEAN = ∅

EQUAL VARIANCE (SMALL)

2- INITIAL WEIGHTS

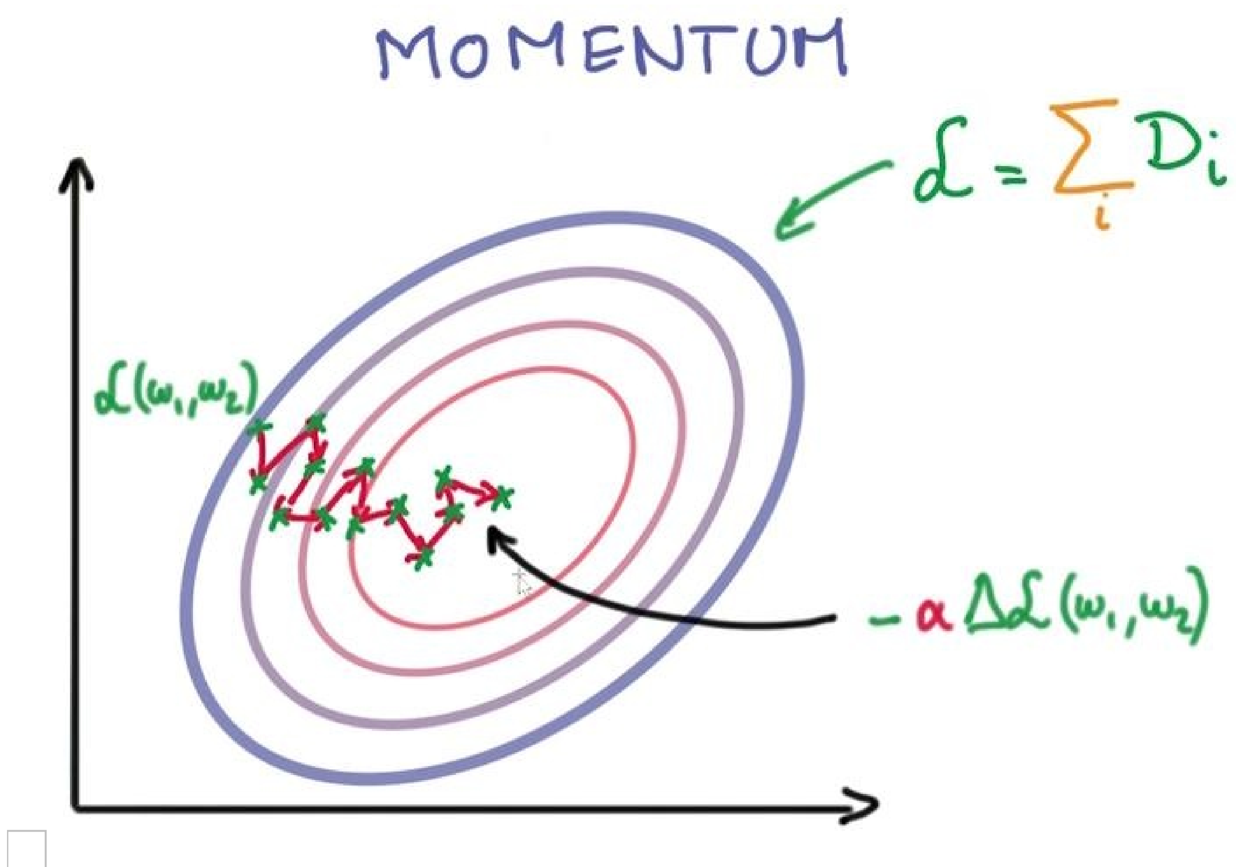
RANDOM!

MEAN = ∅

EQUAL VARIANCE (SMALL TOO)

- Momentum

> 考虑以前的平均调整方向来决定每一步的调整方向



- Learning Rate Decay
  - 训练越靠近目标，步长应该越小
- Parameter Hyperspace
  - Learning Rate（即调整的step）不是越大越好，可能有瓶颈
  - SGD有许多参数可以调整，所以被称为黑魔法

MANY HYPER-PARAMETERS
- INITIAL LEARNING RATE
- LEARNING RATE DECAY
- MOMENTUM
- BATCH SIZE
- WEIGHT INITIALIZATION

  - AdaGurad
    - 自动执行momentum和learning rate decay
    - 使得SGD对参数不像原来那样敏感
    - 自动调整效果不如原来的好，但仍然是一个option

扩展阅读：SGD

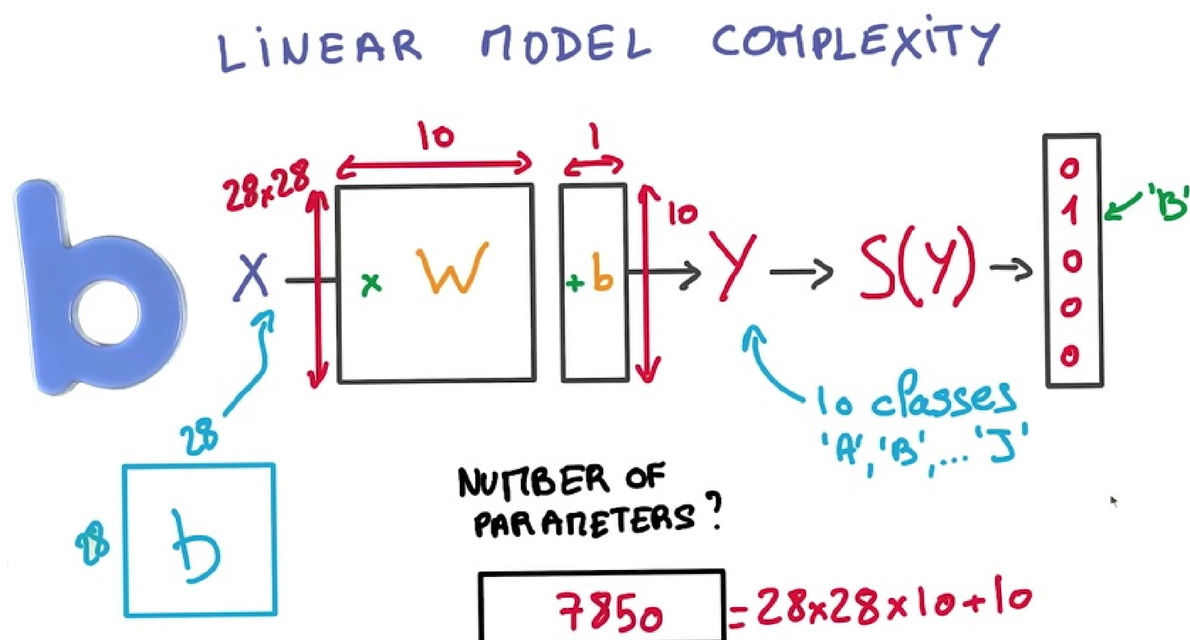
> 觉得得我的文章对您有帮助的话，就给个star吧～
>
> - SGD有许多参数可以调整，所以被称为黑魔法阅读
> - SGD有许多参数可以调整，所以被称为黑魔法

Deep Neural Network

- Limit of Linear Model
- Neural network
  - 神经网络实践
- 优化神经网络：Deep Network
  - 防止深度神经网络过拟合
    - Regularization
    - Dropout
  - 深度神经网络实践

Limit of Linear Model

- 实际要调整的参数很多



> 如果有N个Class，K个Label，需要调整的参数就有(N+1)K个

- Linear Model不能应对非线性的问题

LINEAR MODELS ARE ... LINEAR!

$$Y = X_1 + X_2$$ ✓

$$Y = X_1 \times X_2$$ ✗

- Linear Model的好处
    - GPU就是设计用于大矩阵相乘的，因此它们用来计算Linear Model非常高效
    - Stable：input的微小改变不会很大地影响output

$$Y = WX \rightarrow \Delta Y \sim |W| \Delta X$$

SMALL　BOUNDED　SMALL

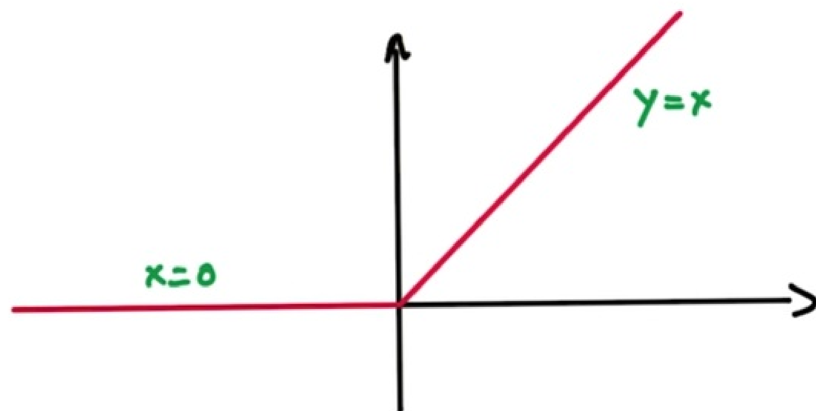
$$\frac{\Delta Y}{\Delta X} = W^T$$

CONSTANTS

$$\frac{\Delta Y}{\Delta W} = X^T$$

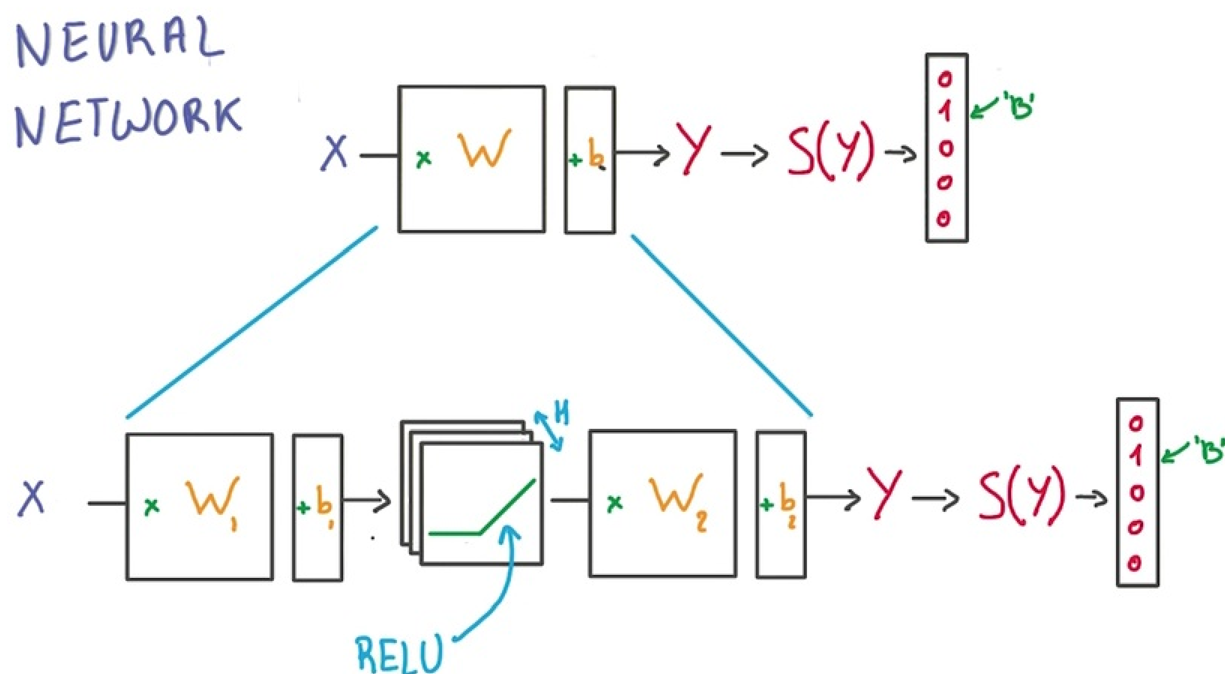
  - 求导方便：线性求导是常数
- 我们想要参数函数是线性的，但整个model是非线性的
- 所以需要对各个线性模型做非线性组合
  - 最简单的非线性组合：分段线性函数（RELU）

RECTIFIED LINEAR UNITS (RELU)



Neural network

- 用一个RELU作为中介，一个Linear Model的输出作为其输入，其输出作为另一个Linear Model的输入，使其能够解决非线性问题

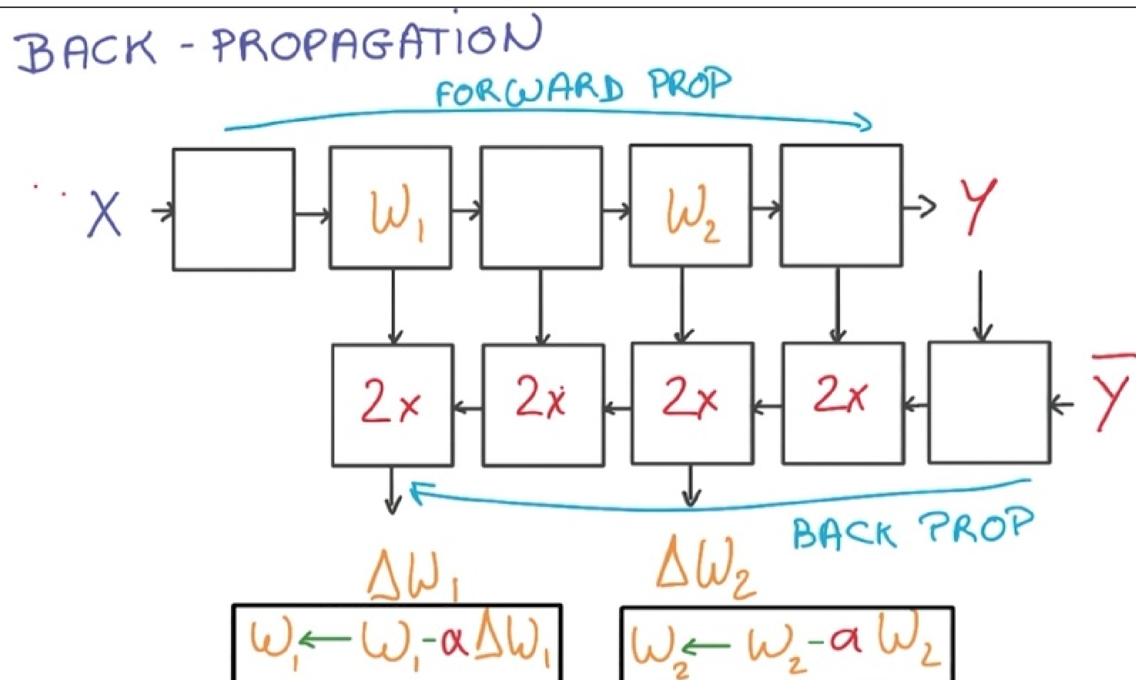


- 神经网络并不一定要完全像神经元那样工作
- Chain Rule：复合函数求导规律

$$[g(f(x))]' = g'(f(x)) \cdot f'(x)$$

DERIVATIVE PRODUCT

- 大量可重用的数据，易于实现（简单的数据流）
- Back propagation



- 计算train_loss时，数据正向流入，计算梯度时，逆向计算
- 计算梯度需要的内存和计算时间是计算train_loss的两倍

- 利用上面的知识，结合lesson1中的SGD，训练一个全连接神经网络：神经网络实践

> 扩展阅读：西瓜书第五章·神经网络

全连接神经网络

辅助阅读：TensorFlow中文社区教程 - 英文官方教程

> 代码见：full_connect.py

Linear Model

- 加载lesson 1中的数据集
- 将Data降维成一维，将label映射为one-hot encoding

  ```
  def reformat(dataset, labels):
    dataset = dataset.reshape((-1, image_size * image_size)).astype(np.float32)
    # Map 0 to [1.0, 0.0, 0.0 ...], 1 to [0.0, 1.0, 0.0 ...]
    labels = (np.arange(num_labels) == labels[:, None]).astype(np.float32)
    return dataset, labels
  ```

TensorFlow Graph

- 使用梯度计算train_loss，用tf.Graph()创建一个计算单元
  - 用tf.constant将dataset和label转为tensorflow可用的训练格式（训练中不可修改）
  - 用tf.truncated_normal生成正太分布的数据，作为W的初始值，初始化b为可变的0矩阵
  - 用tf.variable将上面的矩阵转为tensorflow可用的训练格式（训练中可以修改）
  - TensorFlow的Tutorial里有这样一句话：

  > it is also good practice to initialize them with a slightly positive initial bias to avoid "dead neurons."

  所以init时value设置为0.1会比较好（恩，这也是黑魔法）

  - 用tf.matmul实现矩阵相乘，计算WX+b，这里实际上logit只是一个变量，而非结果
  - 用tf.nn.softmax_cross_entropy_with_logits计算WX+b的结果相较于原来的label的train_loss，并求均值
  - 使用梯度找到最小train_loss

    ```
    optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)
    ```

  - 计算相对valid_dataset和test_dataset对应的label的train_loss

  > 上面这些变量都是一种Tensor的概念，它们是一个个的计算单元，我们在Graph中设置了这些计算单元，规定了它们的组合方式，就好像把一个个门电路串起来那样

TensorFLow Session

Session用来执行Graph里规定的计算，就好像给一个个门电路通上电，我们在Session里，给计算单元冲上数据，That's Flow.

- 重复计算单元反复训练800次，提高其准确度
    - 为了快速查看训练效果，每轮训练只给10000个训练数据(subset)，恩，每次都是相同的训练数据
    - 将计算单元graph传给session
    - 初始化参数
    - 传给session优化器 - train_loss的梯度optimizer，训练损失 - train_loss，每次的预测结果，循环执行训练

    ```
    with tf.Session(graph=graph) as session:
        tf.global_variables_initializer().run()
        for step in range(num_steps):
            _, l, predictions = session.run([optimizer, loss, train_prediction])
    ```

    - 值得注意的是，如果session.run的时候，不把optimizer fetch回来，optimizer是不会工作的
    - 在循环过程中，W和b会保留，并不断得到修正
    - 在每100次循环后，会用验证集进行验证一次，验证也同时修正了一部分参数

    ```
    valid_prediction.eval()
    ```

    - 最后用测试集进行测试
    - 注意如果lesson 1中没有对数据进行乱序化，可能训练集预测准确度很高，验证集和测试集准确度会很低

> 这样训练的准确度为83.2%

SGD

- 每次只取一小部分数据做训练，计算loss时，也只取一小部分数据计算loss
    - 对应到程序中，即修改计算单元中的训练数据，
        - 每次输入的训练数据只有128个，随机取起点，取连续128个数据：

        ```
        offset = (step * batch_size) % (train_labels.shape[0] - batch_size)
        batch_data = train_dataset[offset:(offset + batch_size), :]
        batch_labels = train_labels[offset:(offset + batch_size), :]
        ```

    - 由于这里的数据是会变化的，因此用tf.placeholder来存放这块空间

    ```
    tf_train_dataset = tf.placeholder(tf.float32,
                                      shape=(batch_size, image_size * image_s
    ize))
    tf_train_labels = tf.placeholder(tf.float32, shape=(batch_size, num_labels))
    ```

- 计算3000次，训练总数据量为384000，比之前8000000少

> 准确率提高到86.5%，而且准确率随训练次数增加而提高的速度变快了

神经网络

- 上面SGD的模型只有一层WX+b，现在使用一个RELU作为中间的隐藏层，连接两个WX+b
  - 仍然只需要修改Graph计算单元为

    ```
    Y = W2 * RELU(W1*X + b1) + b2
    ```

  - 为了在数学上满足矩阵运算，我们需要这样的矩阵运算：

    ```
    [n * 10] = RELU([n * 784] · [784 * N] + [n * N]) · [N * 10] + [n * 10]
    ```

  - 这里N取1024，即1024个隐藏结点
  - 于是四个参数被修改

    ```
    weights1 = tf.Variable(
            tf.truncated_normal([image_size * image_size, hidden_node_count]))
    biases1 = tf.Variable(tf.zeros([hidden_node_count]))
    weights2 = tf.Variable(
            tf.truncated_normal([hidden_node_count, num_labels]))
    biases2 = tf.Variable(tf.zeros([num_labels]))
    ```

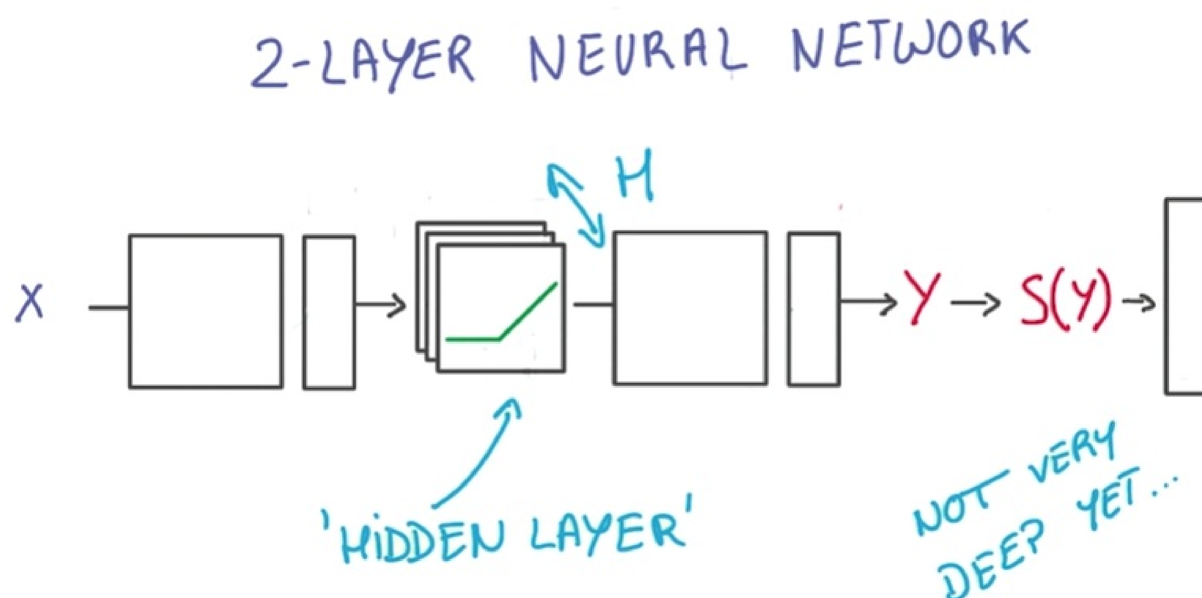
  - 预测值计算方法改为

    ```
    ys = tf.matmul(tf_train_dataset, weights1) + biases1
    hidden = tf.nn.relu(ys)
    logits = tf.matmul(hidden, weights2) + biases2
    ```

  - 计算3000次，可以发现准确率一开始提高得很快，后面提高速度变缓，最终测试准确率提高到88.8%

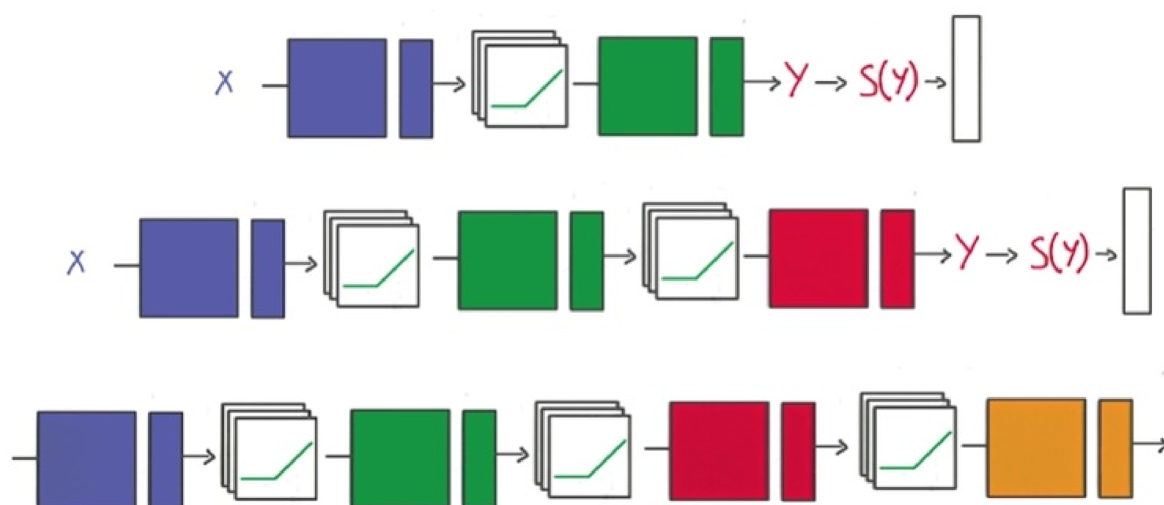
Deep Neural Network

Current two layer neural network:



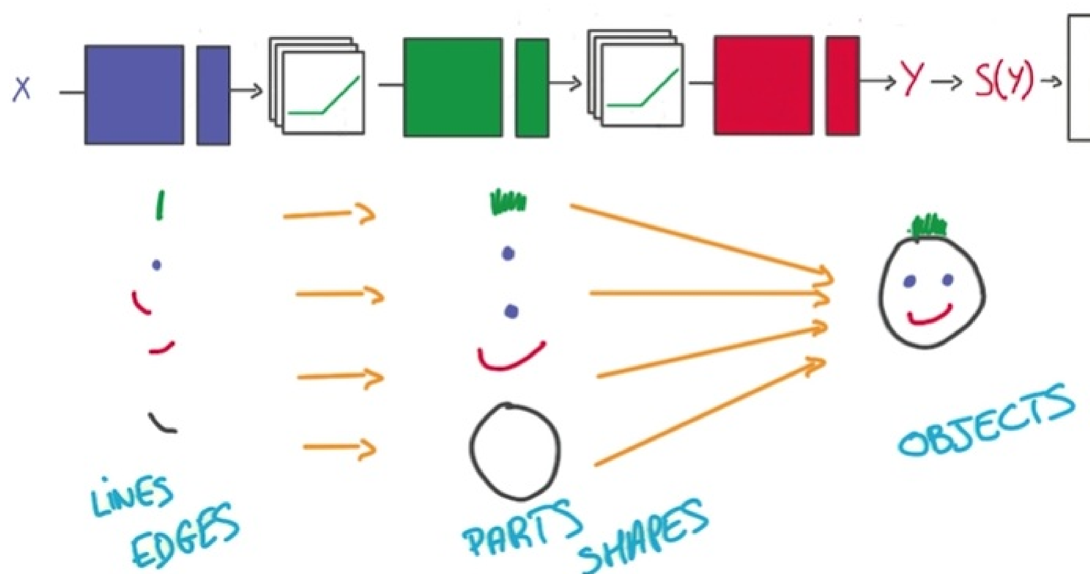
优化：

- 优化RELU(隐藏层), wider
- 增加linear层，layer deeper



  - Performance: few parameters by deeper

- 随层级变高，获得的信息越综合，越符合目标



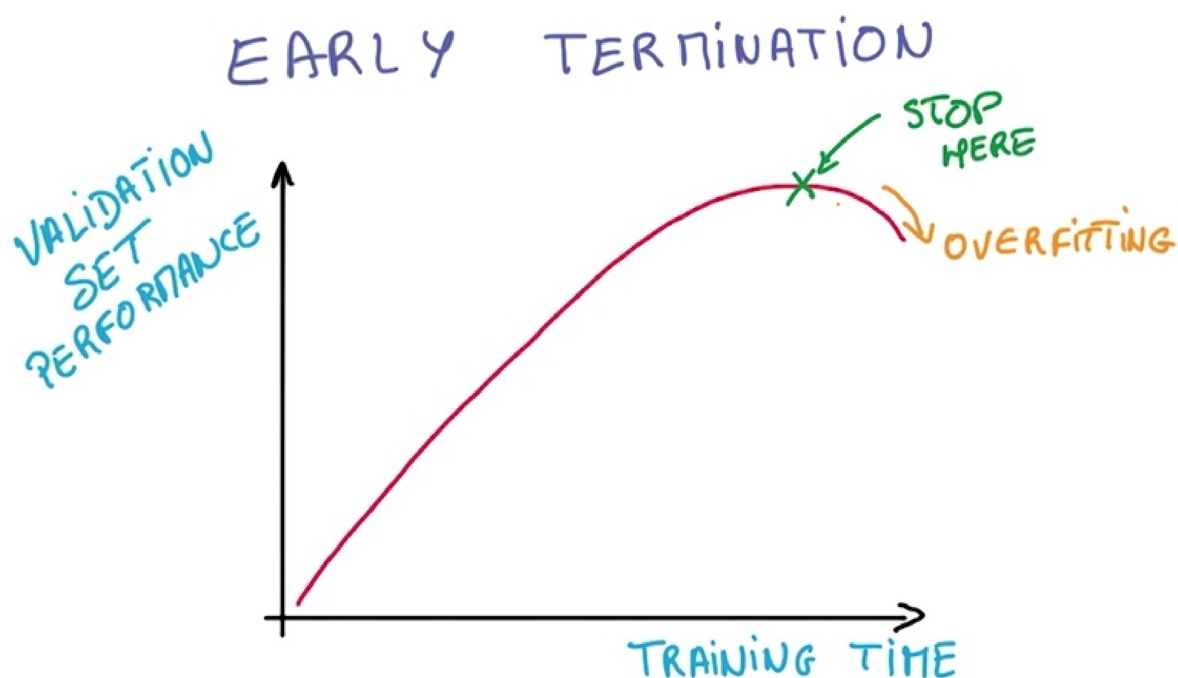
About t-model

- t-model只有在有大量数据时有效
- 今天我们才有高效的大数据训练方法：Better Regularization
- 难以决定适应问题的神经网络的规模，因此通常选择更大的规模，并防止过拟合

Avoid Overfit

Early Termination

- 当训练结果与验证集符合度下降时，就停止训练



Regularization

- 给神经网络里加一些常量，做一些限制，减少自由的参数
- L2 regularization

L2 REGULARIZATION

NEW LOSS

$$\mathcal{L}' = \mathcal{L} + \beta \frac{1}{2}\|W\|_2^2$$

LOSS

$\frac{1}{2}\|W\|_2^2$

W

在计算train loss时，增加一个l2 norm作为新的损失，这里需要乘一个β（Hyper parameter），调整这个新的项的值

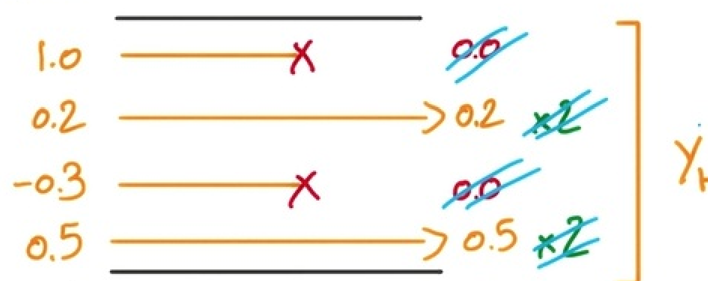
> Hyper parameter：拍脑袋参数→_→

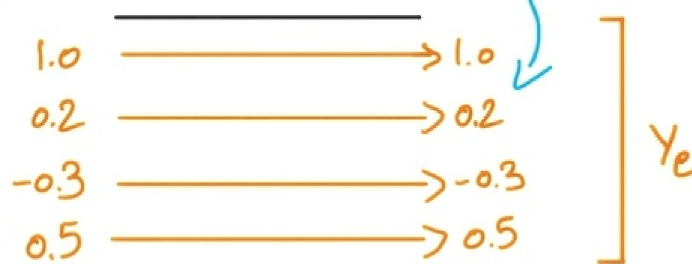
- l2模的导数容易计算，即W本身

DropOut

最近才出现，效果极其好

- 从一个layer到另一个layer的value被称为activation
- 将一个layer到另一个layer的value的中，随机地取一半的数据变为0，这其实是将一半的数据直接丢掉
- 由于数据缺失，所以就强迫了神经网络学习redundant的知识，以作为损失部分的补充
- 由于神经网络中总有其他部分作为损失部分的补充，所以最后的结果还是OK的
- More robust and prevent overfit
- 如果这种方法不能生效，那可能就要使用更大的神经网络了
- 评估神经网络时，就不需要DropOut，因为需要确切的结果
- 可以将所有Activation做平均，作为评估的依据
- 因为我们在训练时去掉了一半的随机数据，如果要让得到Activation正确量级的平均值，就需要将没去掉的数据翻倍





深度神经网络实践

代码见nn_overfit.py

优化

Regularization

在前面实现的RELU连接的两层神经网络中，加Regularization进行约束，采用加l2 norm的方法，进行负反馈：

代码实现上，只需要对tf_sgd_relu_nn中train_loss做修改即可：

- 可以用tf.nn.l2_loss(t)对一个Tensor对象求l2 norm
- 需要对我们使用的各个W都做这样的计算（参考tensorflow官方example）

  ```
  l2_loss = tf.nn.l2_loss(weights1) + tf.nn.l2_loss(weights2)
  ```

- 添加到train_loss上
- 这里还有一个重要的点，Hyper Parameter: β
  - 我觉得这是一个拍脑袋参数，取什么值都行，但效果会不同，我这里解释一下我取β=0.001的理由
  - 如果直接将l2_loss加到train_loss上，每次的train_loss都特别大，几乎只取决于l2_loss
  - 为了让原本的train_loss与l2_loss都能较好地对参数调整方向起作用，它们应当至少在同一个量级

- 观察不加l2_loss，step 0 时，train_loss在300左右
- 加l2_loss后， step 0 时，train_loss在300000左右
- 因此给l2_loss乘0.0001使之降到同一个量级

```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
labels=tf_train_labels)) + 0.001 * l2_loss
```

- 所有其他参数不变，训练3000次，准确率提高到92.7%
- 黑魔法之所以为黑魔法就在于，这个参数可以很容易地影响准确率，如果β = 0.002，准确率提高到93.5%

OverFit问题

在训练数据很少的时候，会出现训练结果准确率高，但测试结果准确率低的情况

- 缩小训练数据范围：将把batch数据的起点offset的可选范围变小（只能选择0-1128之间的数据）：

```
offset_range = 1000
offset = (step * batch_size) % offset_range
```

- 可以看到，在step500后，训练集就一直是100%，验证集一直是77.6%，准确度无法随训练次数上升，最后的测试准确度是85.4%

DropOut

采取Dropout方式强迫神经网络学习更多知识

> 参考aymericdamien/TensorFlow-Examples中dropout的使用

- 我们需要丢掉RELU出来的部分结果
- 调用tf.nn.dropout达到我们的目的：

```
keep_prob = tf.placeholder(tf.float32)
if drop_out:
  hidden_drop = tf.nn.dropout(hidden, keep_prob)
  h_fc = hidden_drop
```

- 这里的keep_prob是保留概率，即我们要保留的RELU的结果所占比例，tensorflow建议的语法是，让它作为一个placeholder，在run时传入
- 当然我们也可以不用placeholder，直接传一个0.5：

```
if drop_out:
  hidden_drop = tf.nn.dropout(hidden, 0.5)
  h_fc = hidden_drop
```

- 这种训练的结果就是，虽然在step 500对训练集预测没能达到100%（起步慢），但训练

集预测率达到100%后，验证集的预测正确率仍然在上升

- 这就是Dropout的好处，每次丢掉随机的数据，让神经网络每次都学习到更多，但也需要知道，这种方式只在我们有的训练数据比较少时很有效
- 最后预测准确率为88.0%

Learning Rate Decay

随着训练次数增加，自动调整步长

- 在之前单纯两层神经网络基础上，添加Learning Rate Decay算法
- 使用tf.train.exponential_decay方法，指数下降调整步长，具体使用方法官方文档说的特别清楚
- 注意这里面的cur_step传给优化器，优化器在训练中对其做自增计数
- 与之前单纯两层神经网络对比，准确率直接提高到90.6%

Deep Network

增加神经网络层数，增加训练次数到20000

- 为了避免修改网络层数需要重写代码，用循环实现中间层

```
# middle layer
for i in range(layer_cnt - 2):
    y1 = tf.matmul(hidden_drop, weights[i]) + biases[i]
    hidden_drop = tf.nn.relu(y1)
    if drop_out:
        keep_prob += 0.5 * i / (layer_cnt + 1)
        hidden_drop = tf.nn.dropout(hidden_drop, keep_prob)
```

- 初始化weight在迭代中使用

```
for i in range(layer_cnt - 2):
    if hidden_cur_cnt > 2:
        hidden_next_cnt = int(hidden_cur_cnt / 2)
    else:
        hidden_next_cnt = 2
    hidden_stddev = np.sqrt(2.0 / hidden_cur_cnt)
    weights.append(tf.Variable(tf.truncated_normal([hidden_cur_cnt, hidden_next_cnt
], stddev=hidden_stddev)))
    biases.append(tf.Variable(tf.zeros([hidden_next_cnt])))
    hidden_cur_cnt = hidden_next_cnt
```

  - 第一次测试时，用正太分布设置所有W的数值，将标准差设置为1，由于网络增加了一层，寻找step调整方向时具有更大的不确定性，很容易导致loss变得很大
  - 因此需要用stddev调整其标准差到一个较小的范围（怎么调整有许多研究，这里直接找了一个来用）

```
stddev = np.sqrt(2.0 / n)
```

- 启用regular时，也要适当调一下β，不要让它对原本的loss造成过大的影响
- DropOut时，因为后面的layer得到的信息越重要，需要动态调整丢弃的比例，到后面的layer，丢弃的比例要减小

```
keep_prob += 0.5 * i / (layer_cnt + 1)
```

- 训练时，调节参数，你可能遇到消失的梯度问题， 对于一个幅度为1的信号，在BP反向传播梯度时，每隔一层下降0.25，指数下降使得后面的层级根本接收不到有效的训练信号
- 官方教程表示最好的训练结果是，准确率97.5%，
- 我的nn_overfit.py开启六层神经网络， 启用Regularization、DropOut、Learning Rate Decay， 训练次数20000（应该还有再训练的希望，在这里虽然loss下降很慢了，但仍然在下降），训练结果是，准确率95.2%

神经网络做数据分类

问题描述

给定两个范围在[-1, 1]之间的数字[x1, x2]，求落在[-0.5, 0.5]之间的数字个数

思路

- 构建一个神经网络，包含若干层，将一系列训练数据输入训练参数进行预测
- 神经网络的每层用一个Relu(Wx+b)实现

问题分解

- 构建数据集
  - 实际上，我们需要先判断x是否落在目标区间，是，则为1，不是则为0，将结果相加
  - 代码见train_data函数
- 神经网络搭建
  - 思路同之前的深度神经网络训练手写文字识别
  - 暂时不做优化
  - 仅搭建两层神经网络
  - 暂时不做validate，因为数据充分，每次训练都是新数据，新数据都相当于validate

> 代码：digit_nn.py

训练结果

仅截取最后一次结果为例

```
current first data [0.206416, 0.101028]
current first predict: [0.000000, 0.000177, 0.999823]
Minibatch loss at step 9980: 0.036539
Minibatch accuracy: 100.0%
```

Convolutional Networks

> deep dive into images and convolutional models

Convnet

BackGround

- 人眼在识别图像时，往往从局部到全局
- 局部与局部之间联系往往不太紧密
- 我们不需要神经网络中的每个结点都掌握全局的知识，因此可以从这里减少需要学习的参数数量

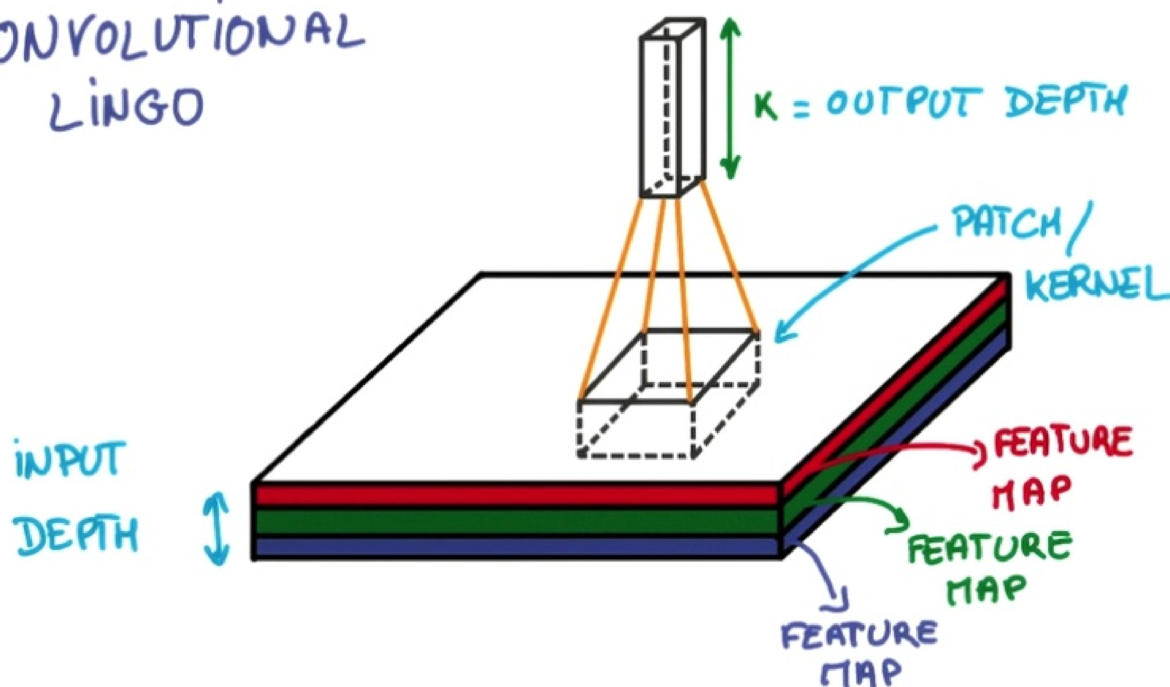
Weight share

- 但这样参数其实还是挺多的，所以有了另一种方法：权值共享

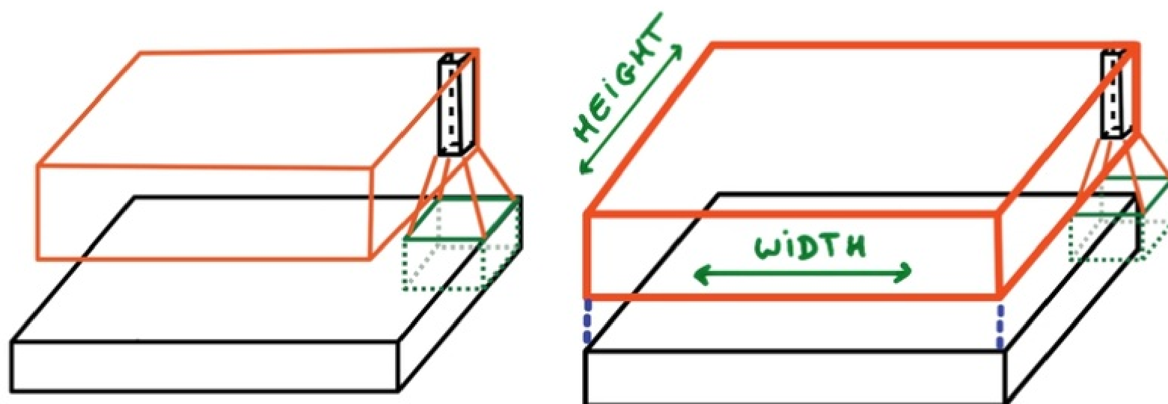
> Share Parameters across space

- 取图片的一小块，在上面做神经网络分析，会得到一些预测
- 将切片做好的神经网络作用于图片的每个区域，得到一系列输出
- 可以增加切片个数提取更多特征
- 在这个过程中，梯度的计算跟之前是一样的

Concept

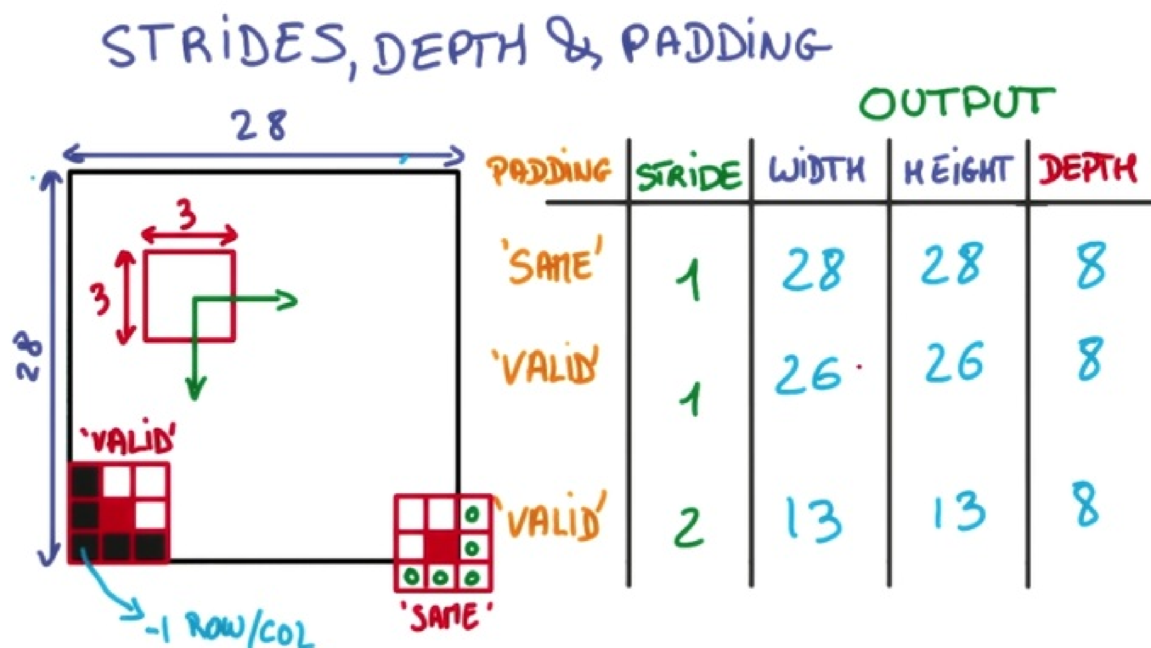


- Patch/Kernel：一个局部切片
- Depth: 数据的深度，图像数据是三维的，长宽和RGB，神经网络的预测输出也属于一维
- Feature Map：每层Conv网络，因为它们将前一层的feature映射到后一层（Output map）



- Stride: 移动切片的步长，影响取样的数量
- 在边缘上的取样影响Conv层的面积，由于移动步长不一定能整除整张图的像素宽度，不越过边缘取样会得到Valid Padding， 越过边缘取样会得到Same Padding

- Example



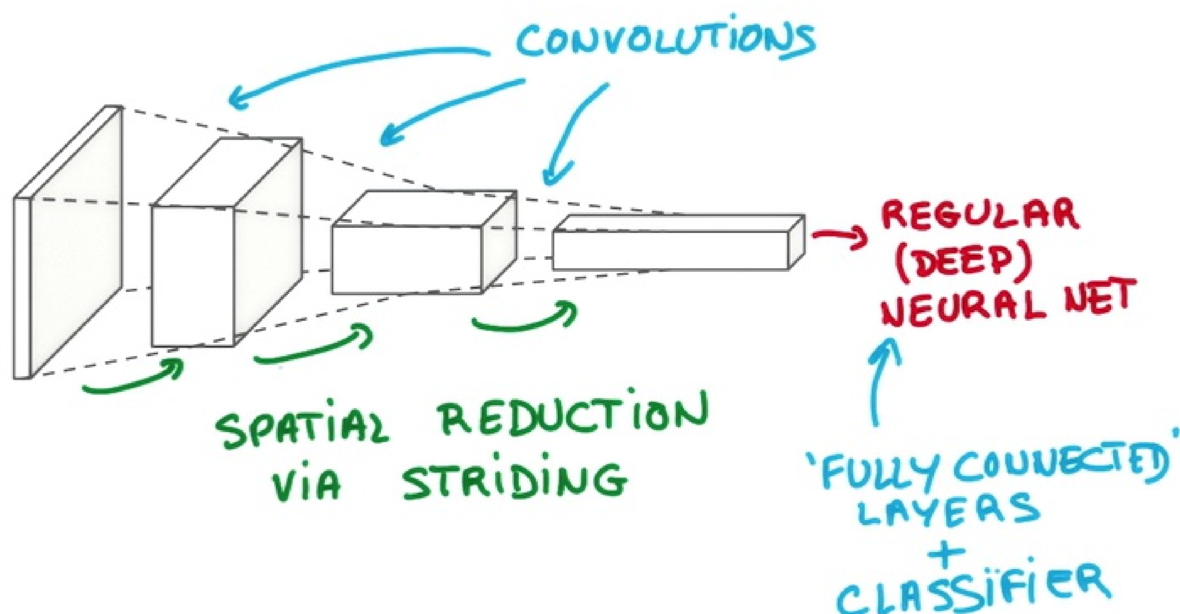
  - 用一个3x3的网格在一个28x28的图像上做切片并移动
  - 移动到边缘上的时候，如果不超出边缘，3x3的中心就到不了边界
  - 因此得到的内容就会缺乏边界的一圈像素点，只能得到26x26的结果
  - 而可以越过边界的情况下，就可以让3x3的中心到达边界的像素点
  - 超出部分的矩阵补零就行

Deep Convnet

在Convnet上套Convnet，就可以一层一层综合局部得到的信息

OutPut

将一个deep and narrow的feature层作为输入，传给一个Regular神经网络



Optimization

Pooling

将不同Stride的卷积用某种方式合并起来，节省卷积层的空间复杂度。

- Max Pooling 在一个卷积层的输出层上取一个切片，取其中最大值代表这个切片
- 优点
  - 不增加需要调整的参数
  - 通常比其他方法准确
- 缺点：更多Hyper Parameter，包括要取最值的切片大小，以及去切片的步长

> LENET-5, ALEXNET

- Average Pooling 在卷积层输出中，取切片，取平均值代表这个切片

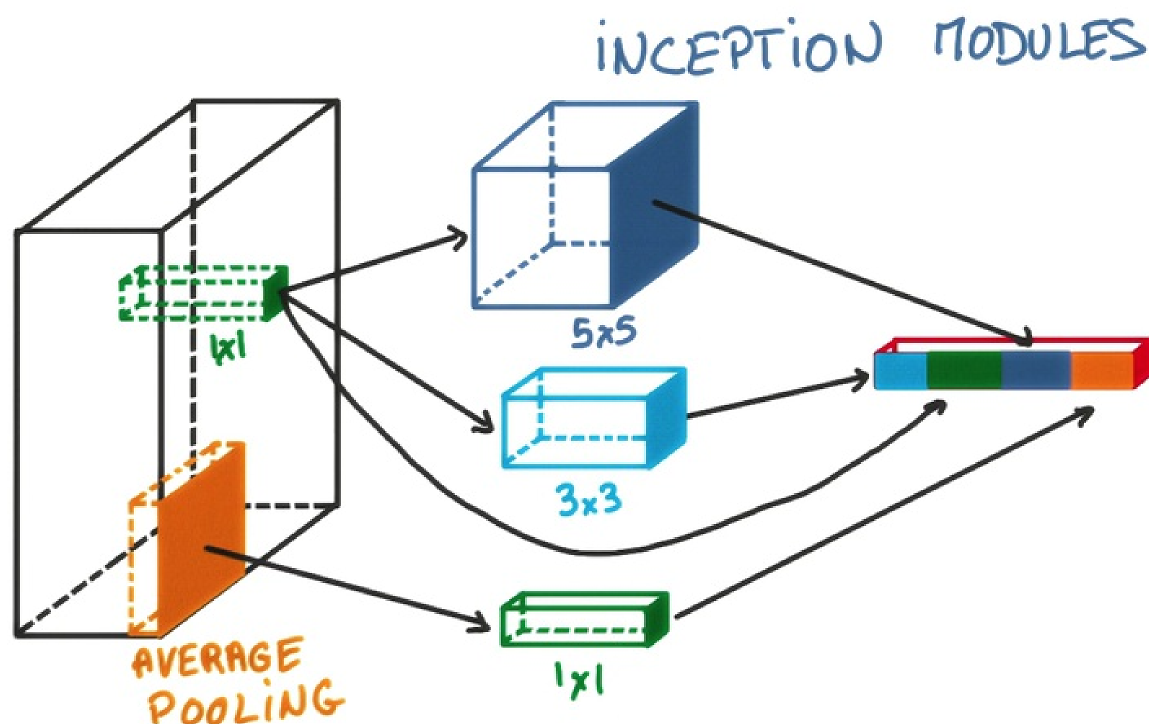
1x1 Convolutions

在一个卷积层的输出层上，加一个1x1的卷积层，这样就形成了一个小型的神经网络。

- cheap for deeper model
- 结合Average Pooling食用效果更加

Inception

对同一个卷积层输出，执行各种二次计算，将各种结果堆叠到新输出的depth方向上



卷积神经网络实践

参考链接

- 张雨石 Conv神经网络
- Bill Xia 卷积神经网络（CNN）

卷积神经网络实践

> 本节介绍如何构造一个简单的CNN模型进行手写数字识别，
>
> 但在现实场景中，往往使用imagenet预训练的深度CNN模型进行迁移学习，能极大地提升预测准确率，
>
> 可参考我在百度大数据竞赛中开源的模型: keras-dog

数据处理

- dataset处理成四维的，label仍然作为one-hot encoding

```
def reformat(dataset, labels, image_size, num_labels, num_channels):
  dataset = dataset.reshape(
      (-1, image_size, image_size, num_channels)).astype(np.float32)
  labels = (np.arange(num_labels) == labels[:, None]).astype(np.float32)
  return dataset, labels
```

- 将lesson2的dnn转为cnn很简单，只要把WX+b改为conv2d(X)+b即可
- 关键在于conv2d

`tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_g`

给定四维的 `input` 和 `filter` tensor，计算一个二维卷积

Args:

- **`input`** : A `Tensor` . type必须是以下几种类型之一: `half` , `float32` , `float64` .
- **`filter`** : A `Tensor` . type和 `input` 必须相同
- **`strides`** : A list of `ints` .一维，长度4， 在 `input` 上切片采样时，每个方向上的滑窗步长，必须和format指定的维度同阶
- **`padding`** : A `string` from: `"SAME", "VALID"` . padding 算法的类型
- **`use_cudnn_on_gpu`** : An optional `bool` . Defaults to `True` .
- **`data_format`** : An optional `string` from: `"NHWC", "NCHW"` ， 默认为 `"NHWC"` 。 指定输入输出数据格式，默认格式为"NHWC", 数据按这样的顺序存储：

  ```
  `[batch, in_height, in_width, in_channels]`
  ```

  也可以用这种方式："NCHW", 数据按这样的顺序存储：

  ```
  `[batch, in_channels, in_height, in_width]`
  ```

- `name` : 操作名，可选.

Returns:

A `Tensor` . type与 `input` 相同

Given an input tensor of shape `[batch, in_height, in_width, in_channels]` and a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, out_channels]`

conv2d实际上执行了以下操作：

1. 将filter转为二维矩阵，shape为 `[filter_height * filter_width * in_channels, output_channels]` .
2. 从input tensor中提取image patches，每个patch是一个*virtual* tensor，shape `[batch, out_height, out_width, filter_height * filter_width * in_channels]` .
3. 将每个filter矩阵和image patch向量相乘

具体来讲，当data_format为NHWC时：

```
output[b, i, j, k] =
    sum_{di, dj, q} input[b, strides[1] * i + di, strides[2] * j + dj, q] *
                    filter[di, dj, q, k]
```

input 中的每个patch都作用于filter，每个patch都能获得其他patch对filter的训练 需要满足 `strides[0] = strides[3] = 1` . 大多数水平步长和垂直步长相同的情况下： `strides = [1, stride, stride, 1]` .

- 然后再接一个WX+b连Relu连WX+b的全连接神经网络即可

Max Pooling

在tf.nn.conv2d后面接tf.nn.max_pool，将卷积层输出减小，从而减少要调整的参数

`tf.nn.max_pool(value, ksize, strides, padding, data_format='`

Performs the max pooling on the input.

Args:

- `value` : A 4-D `Tensor` with shape `[batch, height, width, channels]` and type `tf.float32` .
- `ksize` : A list of ints that has length >= 4. 要执行取最值的切片在各个维度上的尺寸
- `strides` : A list of ints that has length >= 4. 取切片的步长

- `padding` : A string, either `'VALID'` or `'SAME'`. padding算法
- `data_format` : A string. 'NHWC' and 'NCHW' are supported.
- `name` : 操作名，可选

Returns:

A `Tensor` with type `tf.float32`. The max pooled output tensor.

优化

仿照lesson2，添加learning rate decay 和 drop out，可以将准确率提高到90.6%

补充

- 最近在用GPU版本的TensorFlow，发现，如果import tensorflow放在代码第一行，运行会报段错误（pycharm debug模式下不会），因此最好在import tensorflow前import numpy或者其他的module

参考链接

- Tensorflow 中 conv2d 都干了啥
- TensorFlow Example

Deep Models for Text and Sequence

Rare Event

与其他机器学习不同，在文本分析里，陌生的东西（rare event）往往是最重要的，而最常见的东西往往是最不重要的。

语法多义性

- 一个东西可能有多个名字，对这种related文本能够做参数共享是最好的
- 需要识别单词，还要识别其关系，就需要过量label数据

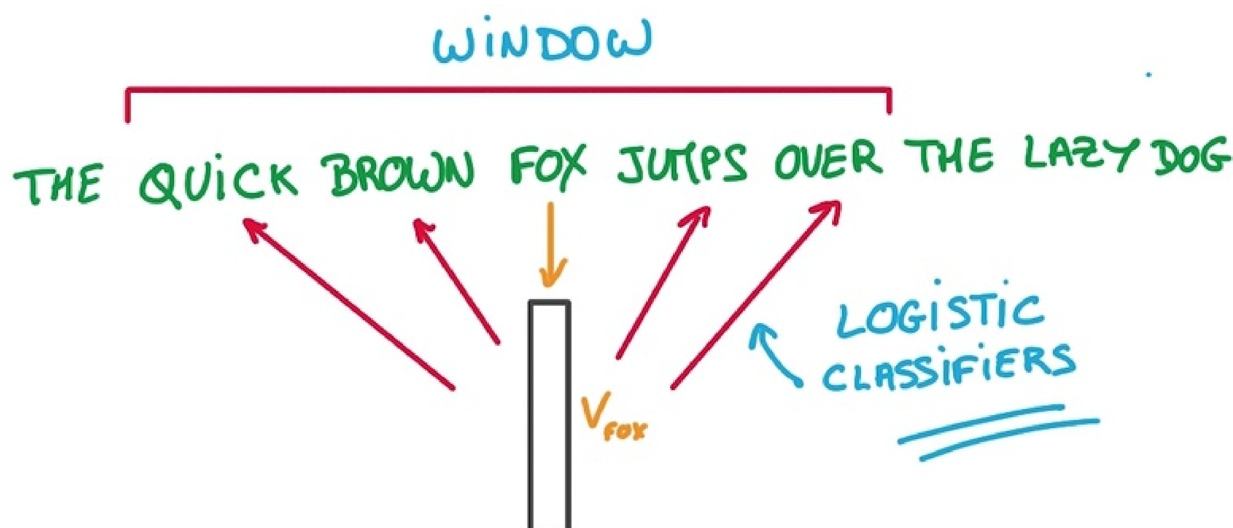
无监督学习

- 不用label进行训练，训练文本是非常多的，关键是要找到训练的内容
- 遵循这样一个思想：相似的词汇出现在相似的场景中
- 不需要知道一个词真实的含义，词的含义由它所处的历史环境决定

Embeddings

- 将单词映射到一个向量（Word2Vec），越相似的单词的向量会越接近
- 新的词可以由语境得到共享参数

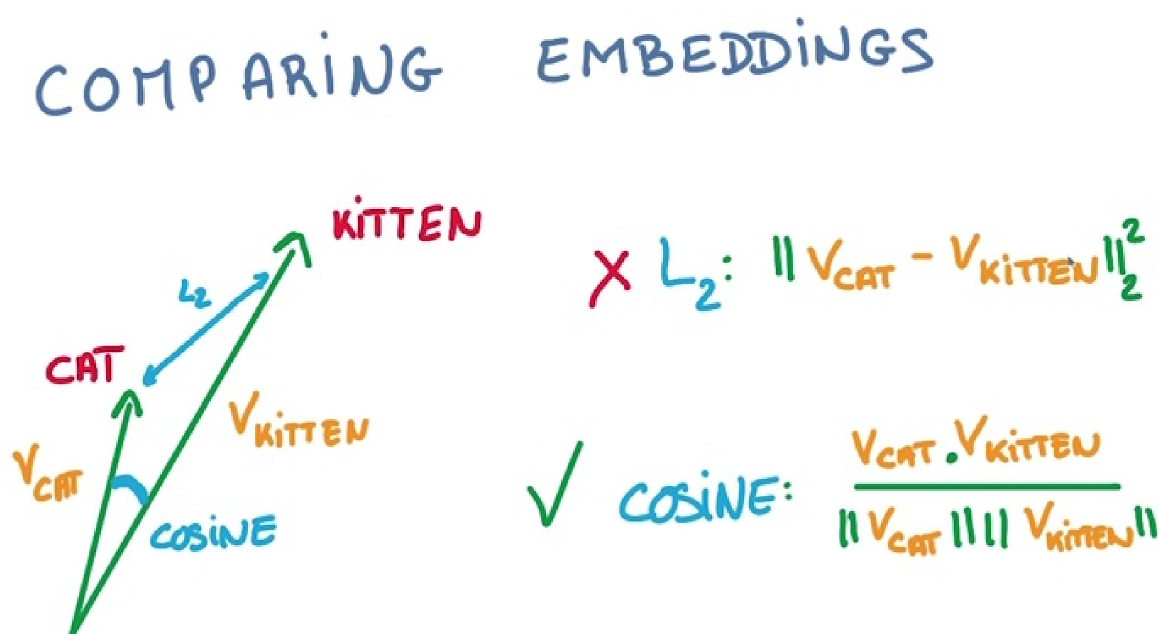
Word2Vec



- 将每个词映射到一个Vector列表(就是一个Embeddings)里，一开始随机，用这个Embedding进行预测
- Context即Vector列表里的邻居
- 目标是让Window里相近的词放在相邻的位置，即预测一个词的邻居
- 用来预测这些相邻位置单词的模型只是一个Logistics Regression， just a simple Linear model

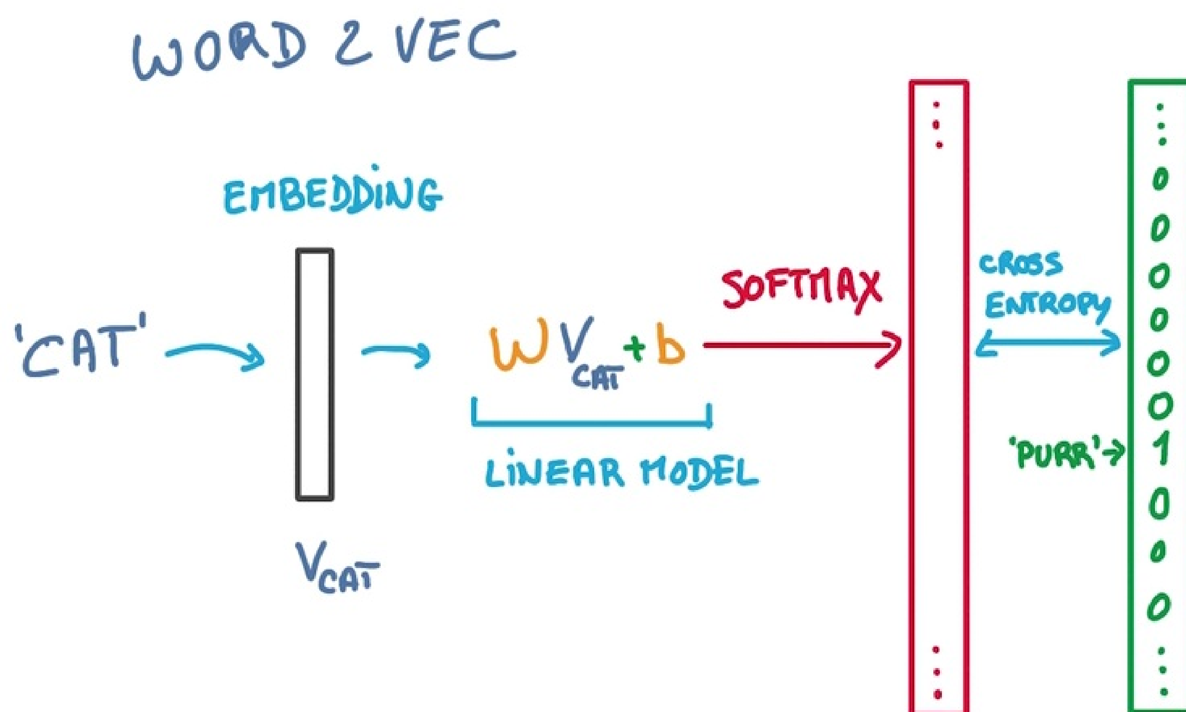
Comparing embeddings

- 比较两个vector之间的夹角大小来判断接近程度，用cos值而非L2计算，因为vector的长度和分类是不相关的：



- 最好将要计算的vector都归一化

Predict Words



- 单词经过embedding变成一个vector
- 然后输入一个WX+b，做一个线性模型
- 输出的label概率为输入文本中的词汇
- 问题在于WX+b输出时，label太多了，计算这种softmax很低效
- 解决方法是，筛掉不可能是目标的label，只计算某个label在某个局部的概率，sample softmax

t-SNE

- 查看某个词在embedding里的最近邻居可以看到单词间的语义接近关系
- 将vector构成的空间降维，可以更高效地查找最近单词，但降维过程中要保持邻居关系（原来接近的降维后还要接近）
- t-SNE就是这样一种有效的方法

类比

- 实际上我们能得到的不仅是单词的邻接关系，由于将单词向量化，可以对单词进行计算
- 可以通过计算进行语义加减，语法加减

WORD ANALOGIES

SEMANTIC ANALOGY

PUPPY → DOG / KITTEN → CAT

SYNTACTIC ANALOGY

TALLER → TALL / SHORTER → SHORT

WORD ANALOGIES

$$V' = V_{PUPPY} - V_{DOG} + V_{CAT}$$



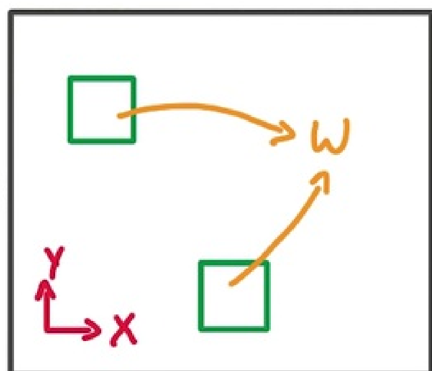
EMBEDDING SPACE

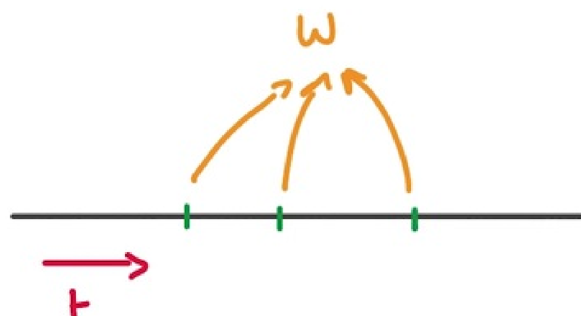
Sequence

文本（Text）是单词（word）的序列，一个关键特点是长度可变，就不能直接变为vector

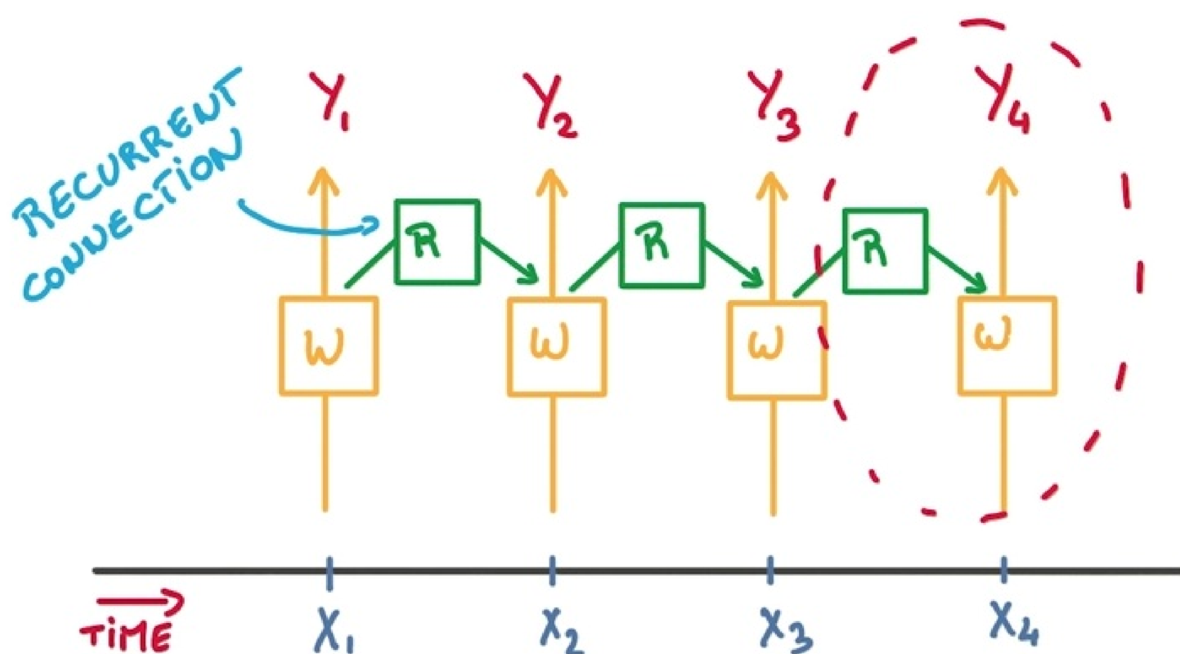
CNN and RNN

CNN 在空间上共享参数，RNN在时间上（顺序上）共享参数



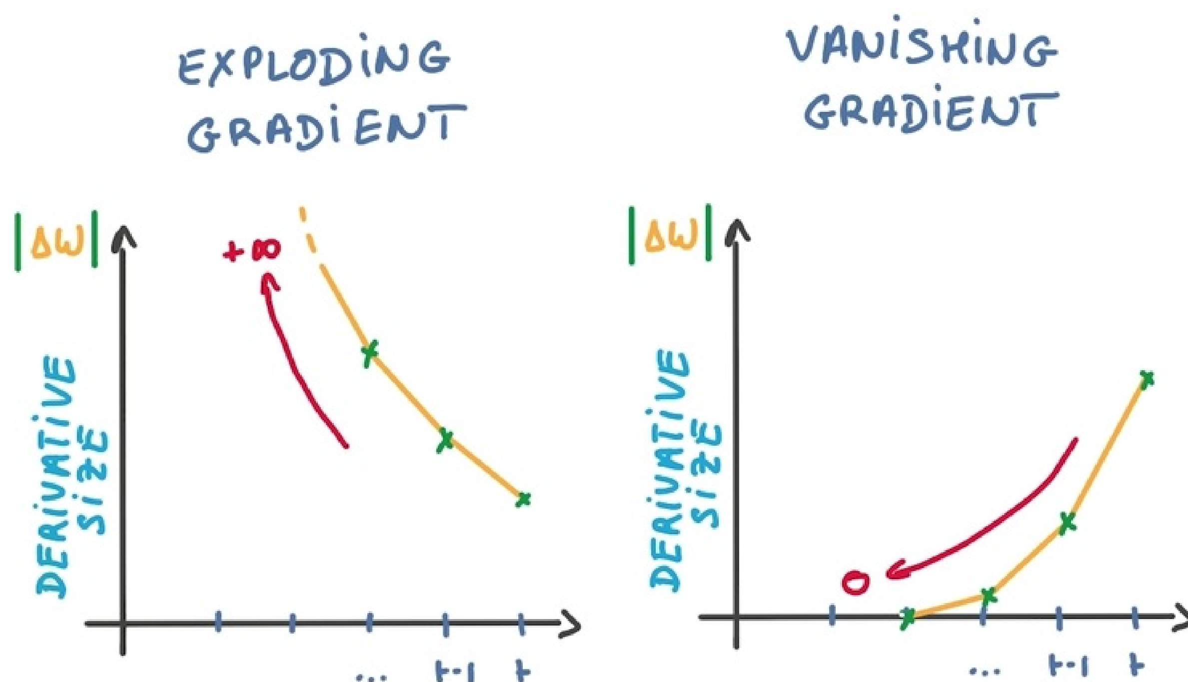


- 在每轮训练中，需要判断至今为之发生了什么，过去输入的所有数据都对当下的分类造成影响
- 一种思路是记忆之前的分类器的状态，在这个基础上训练新的分类器，从而结合历史影响
- 这样需要大量历史分类器
- 重用分类器，只用一个分类器总结状态，其他分类器接受对应时间的训练，然后传递状态



RNN Derivatives

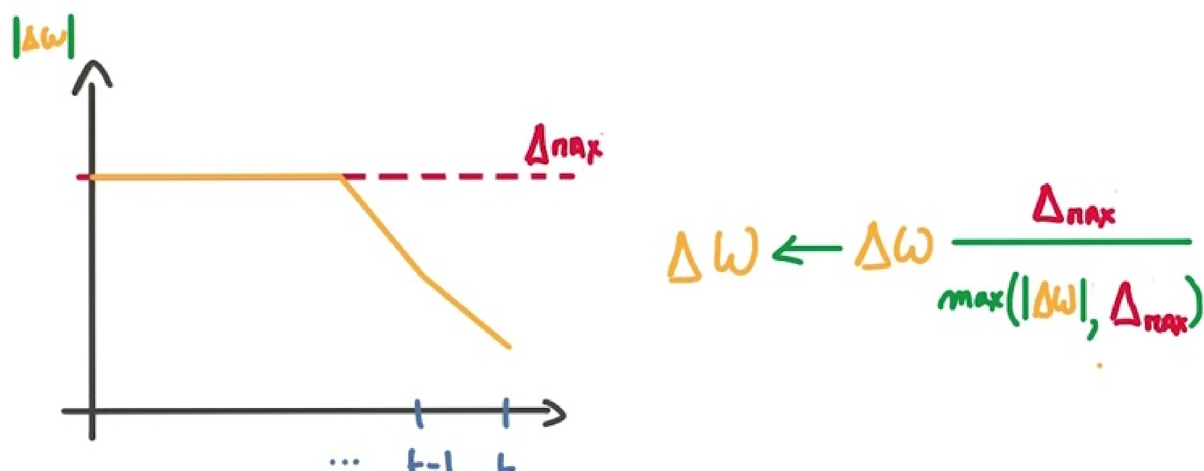
- BackPropagation Through time
- 对同一个weight参数，会有许多求导操作同时更新之
- 对SGD不友好，因为SGD是用许多不相关的求导更新参数，以保证训练的稳定性
- 由于梯度之间的相关性，导致梯度爆炸或者梯度消失



- 使得训练时找不到优化方向，训练失败

Clip Gradient

计算到梯度爆炸的时候，使用一个比值来代替△W（梯度是回流计算的，横坐标从右往左看）

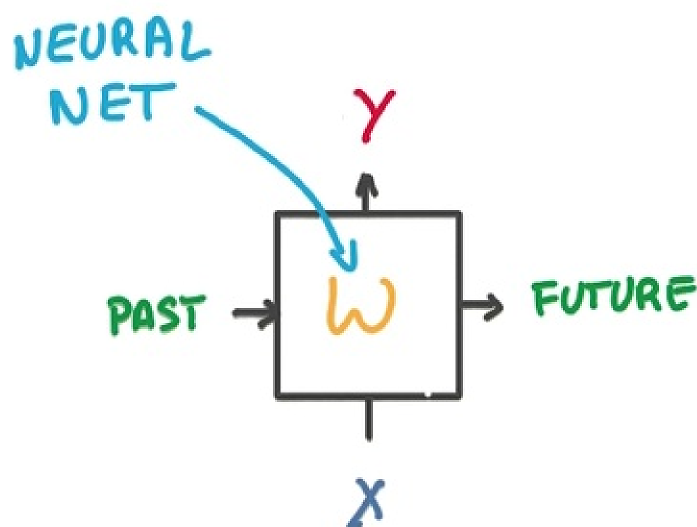


- Hack but cheap and effective

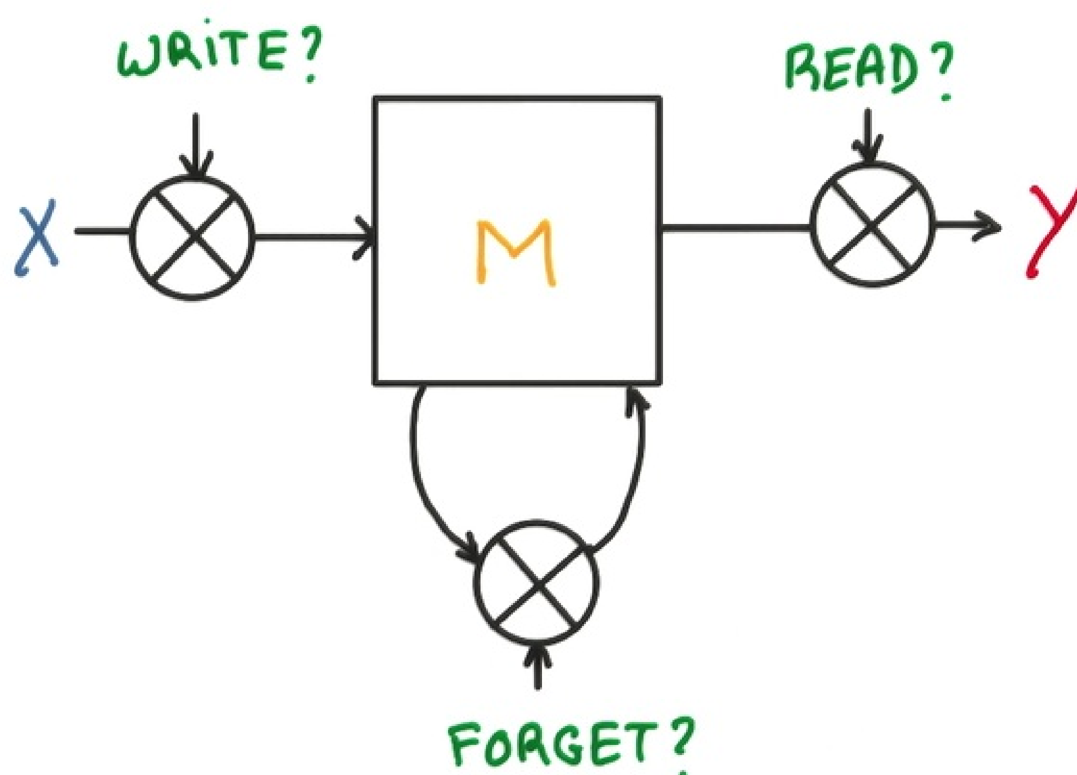
LSTM（Long Short-Term Memory）

梯度消失会导致分类器只对最近的消息的变化有反应，淡化以前训练的参数，也不能用比值的方法来解决

- 一个RNN的model包含两个输入，一个是过去状态，一个是新的数据，两个输出，一个是预测，一个是将来状态

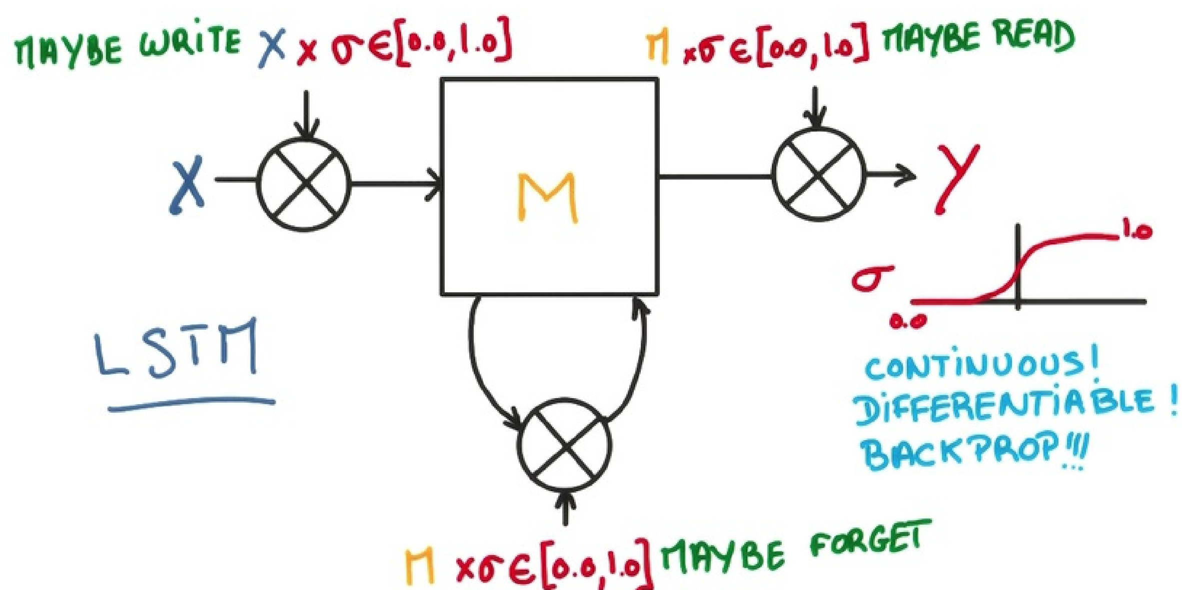


- 中间是一个简单的神经网络
- 将中间的部分换成LSTM-cell就能解决梯度消失问题
- 我们的目的是提高RNN的记忆能力
- Memory Cell

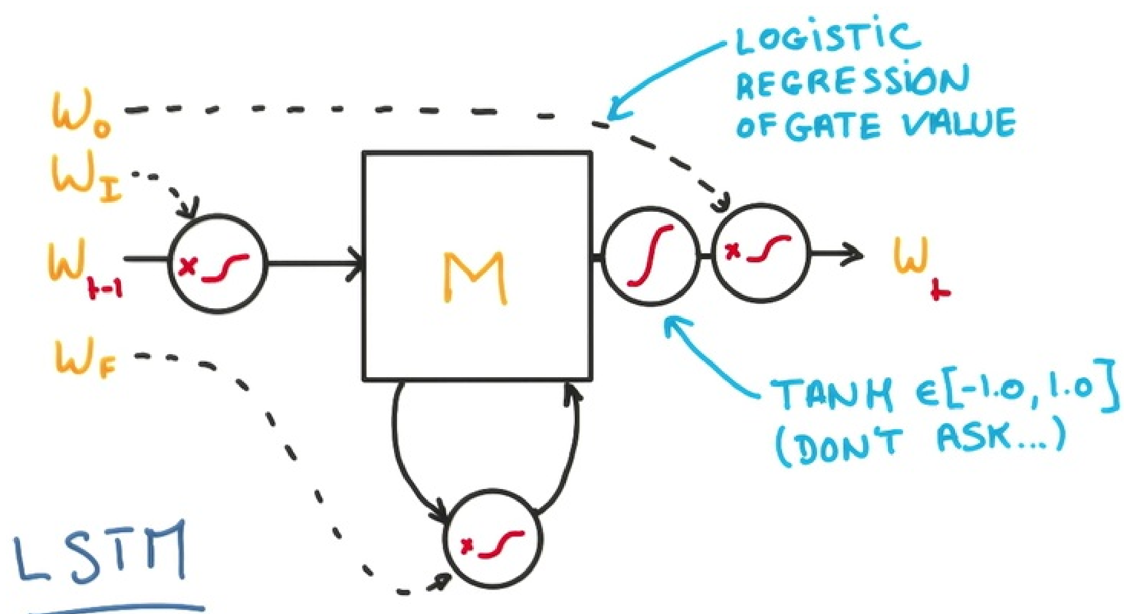


三个门，决定是否写/读/遗忘/写回

- 在每个门上，不单纯做yes/no的判断，而是使用一个权重，决定对输入的接收程度
- 这个权重是一个连续的函数，可以求导，也就可以进行训练，这是LSTM的核心



- 用一个逻辑回归训练这些门，在输出进行归一化



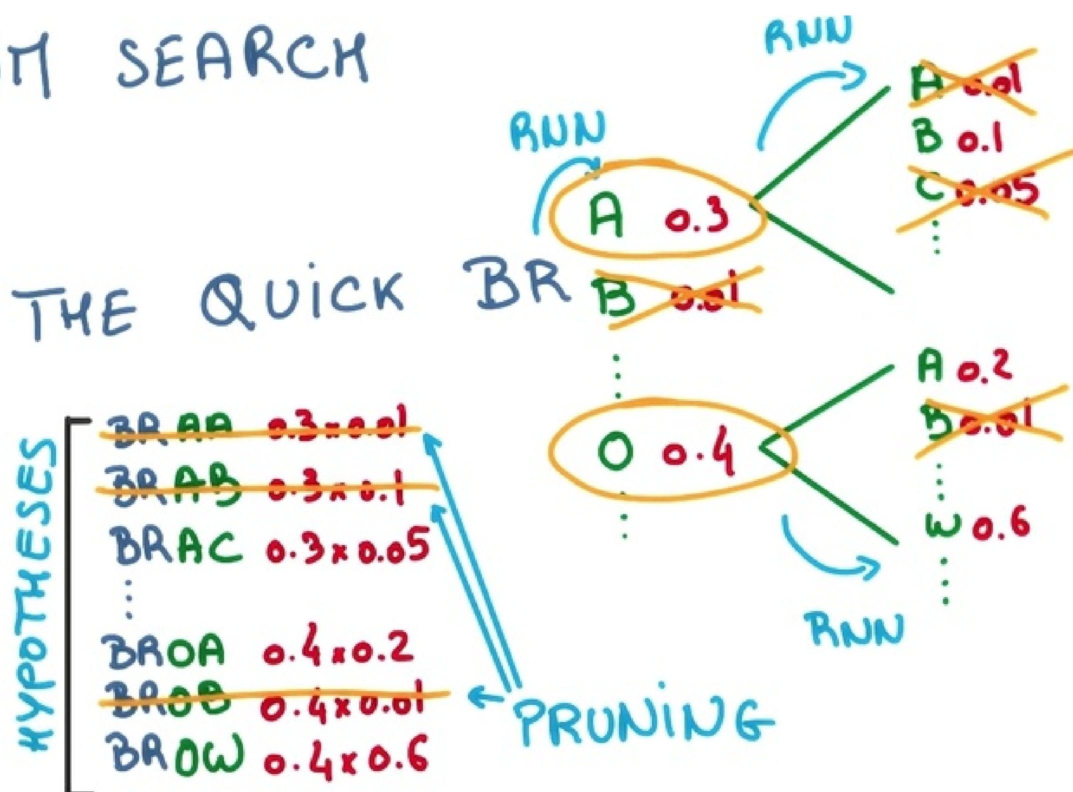
- 这样的模型能让整个cell更好地记忆与遗忘
- 由于整个模型都是线性的，所以可以方便地求导和训练
- 关于lstm有这样一篇博客讲的很好：地址
- 稍微翻了一个中文版

LSTM Regularization

- L2, works
- Dropout on the input or output of data, works

Beam Search

有了上面的模型之后，我们可以根据上文来推测下文，甚至创造下文，预测，筛选最大概率的词，喂回，继续预测......



- 我们可以每次只预测一个字母，but this is greedy，每次都挑最好的那个
- 也可以每次多预测几步，然后挑整体概率较高的那个，以减少偶然因素的影响
- 但这样需要生成的sequence会指数增长
- 因此我们在多预测几步的时候，只为概率比较高的几个候选项做预测，that's beam search.

翻译与识图

- RNN将variable length sequence问题变成了fixed length vector问题，同时因为实际上我们能利用vector进行预测，我们也可以将vector变成sequence

- 我们可以利用这一点，输入一个序列，到一个RNN里，将输出输入到另一个逆RNN序列，形成另一种序列，比如，语言翻译

- 如果我们将CNN的输出接到一个RNN，就可以做一种识图系统

循环神经网络实践

循环神经网络实践

加载数据

- 使用text8作为训练的文本数据集

text8中只包含27种字符：小写的从a到z，以及空格符。如果把它打出来，读起来就像是去掉了所有标点的wikipedia。

- 直接调用lesson1中maybe_download下载text8.zip
- 用zipfile读取zip内容为字符串，并拆分成单词list
- 用connections模块统计单词数量并找出最常见的单词

达成随机取数据的目标

构造计算单元

```
embeddings = tf.Variable(
        tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
```

- 构造一个vocabulary_size x embedding_size的矩阵，作为embeddings容器，
- 有vocabulary_size个容量为embedding_size的向量，每个向量代表一个vocabulary，
- 每个向量的中的分量的值都在-1到1之间随机分布

```
embed = tf.nn.embedding_lookup(embeddings, train_dataset)
```

- 调用tf.nn.embedding_lookup，索引与train_dataset对应的向量，相当于用train_dataset作为一个id，去检索矩阵中与这个id对应的embedding

```
loss = tf.reduce_mean(
        tf.nn.sampled_softmax_loss(softmax_weights, softmax_biases, embed,
                                   train_labels, num_sampled, vocabulary_size))
```

- 采样计算训练损失

```
optimizer = tf.train.AdagradOptimizer(1.0).minimize(loss)
```

- 自适应梯度调节器，调节embedding列表的数据，使得偏差最小

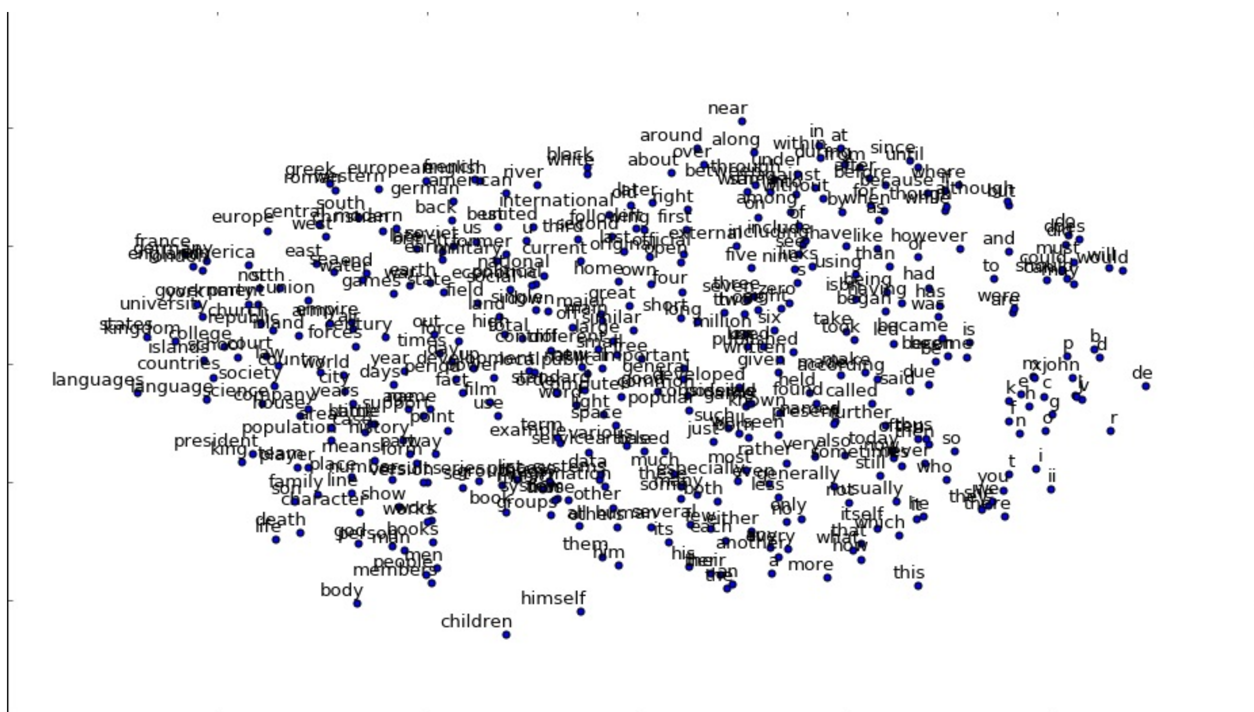
- 预测，并用cos值计算预测向量与实际数据的夹角作为预测准确度（相似度）指标

传入数据进行训练

- 切割数据用于训练，其中：

```
data_index = (data_index + 1) % len(data)
```

- 依旧是每次取一部分随机数据传入
    - 等距离截取一小段文本
    - 构造训练集：每个截取窗口的中间位置作为一个train_data
    - 构造标签：每个截取窗口中，除了train_data之外的部分，随机取几个成为一个list，作为label（这里只随机取了一个）
    - 这样就形成了根据目标词汇预测上下文的机制，即Skip-gram
- 训练100001次，每2000次输出这两千次的平均损失
- 每10000次计算相似度，并输出与验证集中的词最接近的词汇列表
- 用tSNE降维呈现词汇接近程度
- 用matplotlib绘制结果



代码见：word2vec.py

这里我们指定了gpu作为运算设备，会出现这个issue说明的bug，需要进行如下配置解决：

```
config = tf.ConfigProto(allow_soft_placement=True)
session = tf.Session(graph=graph, config=config)
```

CBOW

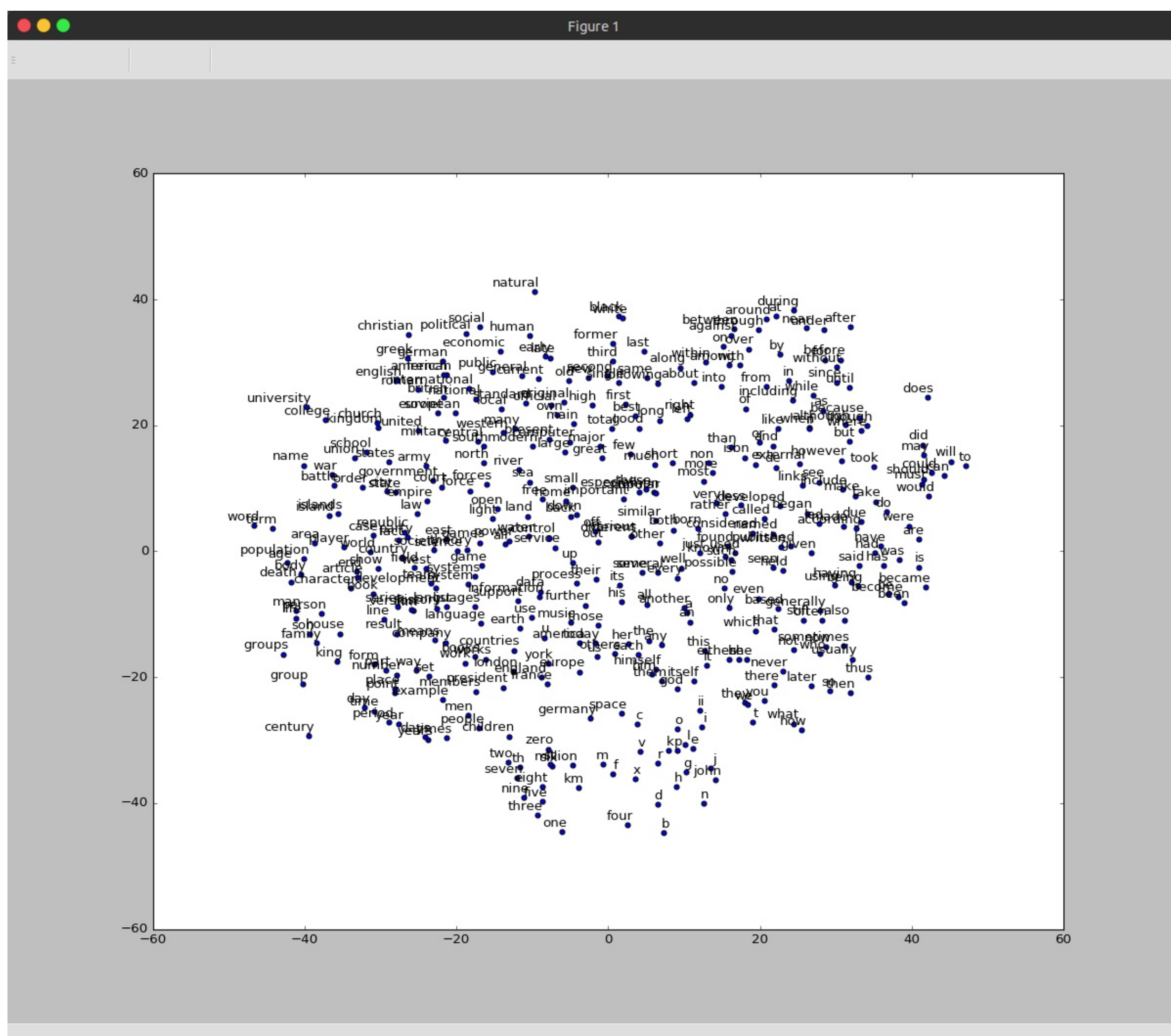
上面训练的是Skip-gram模型，是根据目标词汇预测上下文，而word2vec还有一种方式，CBOW，根据上下文预测目标词汇。

实际上就是将Skip-gram中的输入输出反过来。

- 修改截取数据的方式
  - 构造标签：每个截取窗口的中间位置作为一个train_label
  - 构造训练集：每个截取窗口中，除了train_label之外的部分，作为train_data（这里只随机取了一个）
  - 这样就形成了根据上下文预测目标词汇的机制，即CBOW
- 分别从embeding里找到train_data里每个word对应的vector，用tf.reduce_sum将其相加，将相加结果与train_label比较

```
# Look up embeddings for inputs.
embed = tf.nn.embedding_lookup(embeddings, train_dataset)
# sum up vectors on first dimensions, as context vectors
embed_sum = tf.reduce_sum(embed, 0)
```

- 训练中依旧是调节embeding的参数来优化loss
- 训练结果如下图，可以看到不同单词的接近程度



代码见：cbow.py

RNN 造句

整体思路是，以一个文本中的一个词作为train data，后续的所有词作为train label，从而能够根据一个给定词，预测后续的片段。

训练数据

- BatchGenerator
    - text: 全部的文本数据
    - text_size：全部文本的字符串长度
    - batch_size：每段训练数据的大小
    - num_unrollings：要生成的训练数据段的数目
    - segment：整个训练数据集可以分成几个训练数据片段
    - cursor：重要，

        - 一开始记录每个训练数据片段的起始位置坐标，即这个片段位于text的哪个index
        - 执行next_batch生成一个训练数据的时候，游标会从初始位置自增，直到取够batch_size个数据
    - last_batch：上一个训练数据片段
    - 每调用一次next，生成一个num_unrollings长的array，以last_batch开头，跟着num_unrollings个batch
    - 每个batch的作为train_input，每个batch后面的一个batch作为train_label，每个step训练num_unrolling个batch

lstm-cell

- 为了解决消失的梯度问题，引入lstm-cell，增强model的记忆能力
- 根据这篇论文设计lstm-cell: http://arxiv.org/pdf/1402.1128v1.pdf
- 分别有三个门：输入门，遗忘门，输出门，构成一个cell
    - 输入数据是num_nodes个词，可能有vocabulary_size种词
    - 输入门：

```
input_gate = sigmoid(i * ix + o * im + ib)
```

    - 给输入乘一个vocabulary_size *num_nodes*大小的矩阵，给输出乘一个*num_nodes* num_nodes大小的矩阵;
    - 用这两个矩阵调节对输入数据的取舍程度
    - 用sigmoid这个非线性函数进行激活
    - 遗忘门：

```
forget_gate = sigmoid(i * fx + o * fm + fb)
```

思路同输入门，用以对历史数据做取舍

    - 输出门：

```
output_gate = sigmoid(i * ox + o * om + ob)
```

思路同输入门，用以对输出状态做取舍

    - 组合：

```
update = i * cx + o * cm + cb
state = forget_gate * state + input_gate * tanh(update)
lstm_cell = output_gate * tanh(state)
```

    - 用同样的方式构造新状态update
    - 用遗忘门处理历史状态state

- 用tanh激活新状态update
- 用输入门处理新状态update
- 整合新旧状态，再用tanh激活状态state
- 用输出门处理state

lstm优化

上面的cell中，update，output_gate，forget_gate，input_gate计算方法都是一样的， 可以把四组参数分别合并，一次计算，再分别取出：

```
values = tf.split(tf.matmul(i, input_weights) + tf.matmul(o, output_weights) + bias, g
ate_count, 1)
input_gate = tf.sigmoid(values[0])
forget_gate = tf.sigmoid(values[1])
update = values[2]
```

再将lstm-cell的输出扔到一个WX+b中调整作为输出

代码见：singlew_lstm.py

Optimizer

- 采用one-hot encoding作为label预测
- 采用交叉熵计算损失
- 引入learning rate decay

Flow

- 填入训练数据到placeholder中
- 验证集的准确性用logprob来计算，即对可能性取对数
- 每10次训练随机挑取5个字母作为起始词，进行造句测试
- 你可能注意到输出的sentence是由sample得到的词组成的，而非选择概率最高的词，这是因为，如果一直取概率最高的词，最后会一直重复这个概率最高的词

Beam Search

上面的流程里，每次都是以一个字符作为单位，可以使用多一点的字符做预测，取最高概率的那个，防止特殊情况导致的误判

在这里我们增加字符为2个，形成bigram，代码见：bigram_lstm.py

主要通过BigramBatchGenerator类实现

Embedding look up

由于bigram情况下，vocabulary_size变为 27*27个，使用one-hot encoding 做predict的话会产生非常稀疏的矩阵，浪费算力，计算速度慢

因此引入embedding_lookup,代码见embed_bigram_lstm.py

- 数据输入：BatchGenerator不再生成one-hot-encoding的向量作为输入，而是直接生成bigram对应的index列表
- embedding look up调整embedding，使bigram与vector对应起来
- 将embedding look up的结果喂给lstm cell即可
- 输出时，需要将label和output都转为One-hot-encoding，才能用交叉熵和softmax计算损失
- 在tensor里做data到one-hot-encoding转换时，主要依赖tf.gather函数
- 在对valid数据做转换时，主要依赖one_hot_voc函数

Drop out

- 在lstm cell中对input和output做drop out
- Refer to this article

Seq2Seq

- 最后一个问题是，将一个句子中每个词转为它的逆序字符串，也就是一个seq到seq的转换
- 正经的实现思路是，word 2 vector 2 lstm 2 vector 2 word
- 不过tensorflow已经有了这样一个模型来做这件事情：Seq2SeqModel，关于这个模型可以看这个分析 以及tensorflow的example
- 只需要从batch中，根据字符串逆序的规律生成target sequence，放到seq2seqmodel里即可，主要依赖rev_id函数
- 实现见seq2seq.py
- 注意，用Seq2SeqModel的时候，size和num_layer会在学习到正确的规律前就收敛，我把它调大了一点

```
def create_model(sess, forward_only):
    model = seq2seq_model.Seq2SeqModel(source_vocab_size=vocabulary_size,
                                       target_vocab_size=vocabulary_size,
                                       buckets=[(20, 21)],
                                       size=256,
                                       num_layers=4,
                                       max_gradient_norm=5.0,
                                       batch_size=batch_size,
                                       learning_rate=1.0,
                                       learning_rate_decay_factor=0.9,
                                       use_lstm=True,
                                       forward_only=forward_only)
    return model
```

- 参数含义
    - source_vocab_size: size of the source vocabulary.
    - target_vocab_size: size of the target vocabulary.
    - buckets: a list of pairs (I, O), where I specifies maximum input length that will be processed in that bucket, and O specifies maximum output length. Training instances that have inputs longer than I or outputs longer than O will be pushed to the next bucket and padded accordingly. We assume that the list is sorted, e.g., [(2, 4), (8, 16)].
    - size: number of units in each layer of the model.
    - num_layers: number of layers in the model.
    - max_gradient_norm: gradients will be clipped to maximally this norm.
    - batch_size: the size of the batches used during training; the model construction is independent of batch_size, so it can be changed after initialization if this is convenient, e.g., for decoding.
    - learning_rate: learning rate to start with.
    - learning_rate_decay_factor: decay learning rate by this much when needed.
    - use_lstm: if true, we use LSTM cells instead of GRU cells.
    - num_samples: number of samples for sampled softmax.
    - forward_only: if set, we do not construct the backward pass in the model.

参考链接

- 林洲汉-知乎
- 词向量
- rudolfix - udacity_deeplearn
- Edwardbi - 解析Tensorflow官方English-Franch翻译器demo

理解LSTM 网络

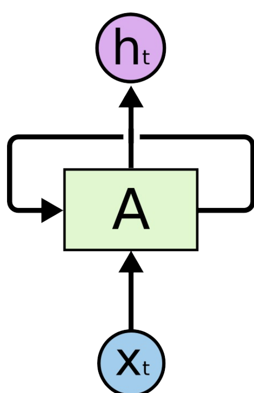
- Posted on August 27, 2015

循环神经网络

人不会每时每刻都从抓取信息这一步开始思考。你在读这篇文章的时候，你对每个次的理解是基于你对以前的词汇的理解的。你不会把所有东西都释放出来然后再从抓取信息开始重新思考，你的思维是有持续性的。

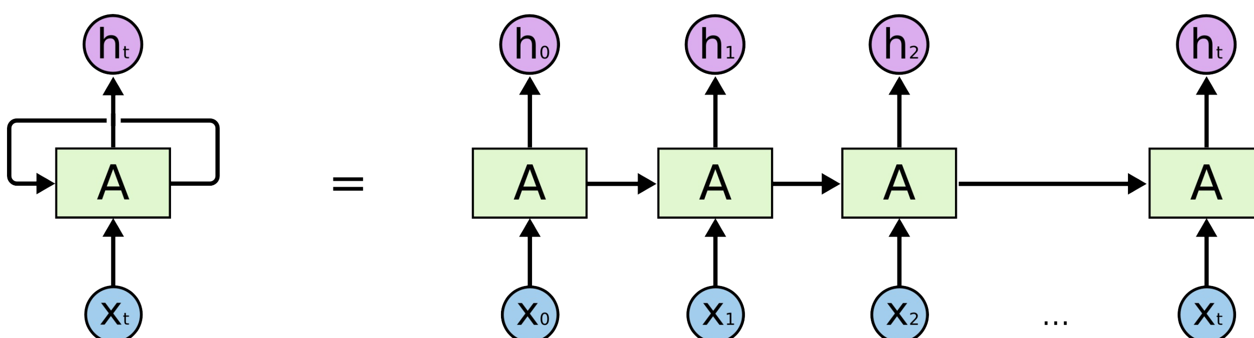
传统的神经网络不能做到这一点， 而且好像这是传统神经网络的一个主要缺点。例如，想象你想要区分一个电影里的每个时刻正在发生的事情。一个传统的神经网络将会如何利用它对过去电影中事件的推理，来预测后续的事件，这个过程是不清晰的。

循环神经网络解决了这个问题。在循环神经网络里，有循环，允许信息持续产生作用。



在上面的图中，一大块神经网络，A，观察一些输入x_t，输出一个值h_t。循环允许信息从网络的一步传到下一步。

这些循环使得循环神经网络似乎有点神秘。然而，如果你想多一点，其实它们跟一个正常的神经网络没有神秘区别。一个循环神经网络可以被认为是同一个网络的多重副本，每个部分会向继任者传递一个信息。想一想，如果我们展开了循环会发生什么：



这个链式本质揭示了，循环神经网络跟序列和列表是紧密相关的。它们是神经网络为这类数据而生的自然架构。

并且它们真的起了作用！在过去的几年里，应用RNN到许多问题中都取得了难以置信的成功：语音识别，语言建模，翻译，图像截取，等等。我会留一个话题，讨论学习Andrej Karpathy的博客能够取得多么令人惊艳的成绩：

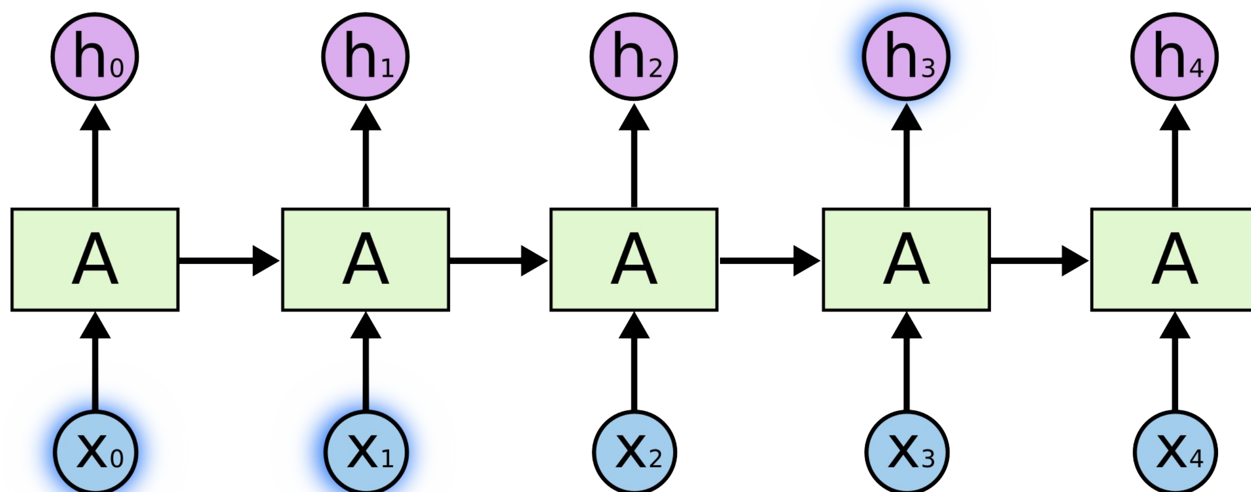
The Unreasonable Effectiveness of Recurrent Neural Networks。但它们真的相当惊艳。

与这些成功紧密相关的是对LSTM的使用，一个非常特殊的循环神经网络的类型。它在许多任务上都能比标准的RNN工作的好得多。几乎所有基于RNN的神经网络取得的激动人心的成果都由LSTM获得。这篇文章将要探索的就是这些LSTM。

Long-Term依赖问题

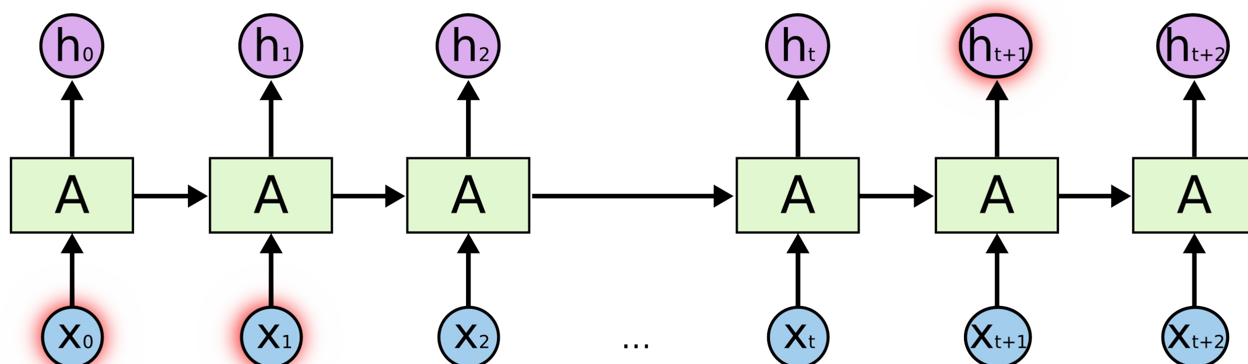
RNN吸引人的一个地方是它们能够链接先前的信息与当前的任务，比如使用先前的视频帧可能预测对于当前帧的理解。如果RNN能够做到这种事情，它们会变得极度有用。但真的可以吗？不好说。

有时候，我们只需要查看最近的信息来执行现在的任务，例如，考虑一个语言模型试图基于先前的词预测下一个词。如果我们要预测“the clouds are in the sky”，我们不需要其他更遥远的上下文 —— 非常明显，下一个词就应该是sky。在这样的例子中，相关信息和目的地之间的距离是很小的。RNN可以学着区使用过去的信息。



但也有一些情况是我们需要更多上下文的。考虑预测这个句子中最后一个词：“I grew up in France... I speak fluent French.” 最近的信息表明下一个词可能是一种语言的名字，但如果我们想要找出是哪种语言，我们需要从更久远的地方获取France的上下文。相关信息和目标之间的距离完全可能是非常巨大的。

不幸的是，随着距离的增大，RNN变得不能够连接信息。



长期依赖导致的神经网络困境

理论上，RNN是绝对能够处理这样的“长期依赖的”。人类可以仔细地从这些词中找到参数然后解决这种形式的一些雏形问题。然而，实践中，RNN似乎不能够学习到这些。 Hochreiter (1991) [German] 和 Bengio, et al. 1994年曾探索过这个问题，他们发现了一些非常根本的导致RNN难以生效的原因。

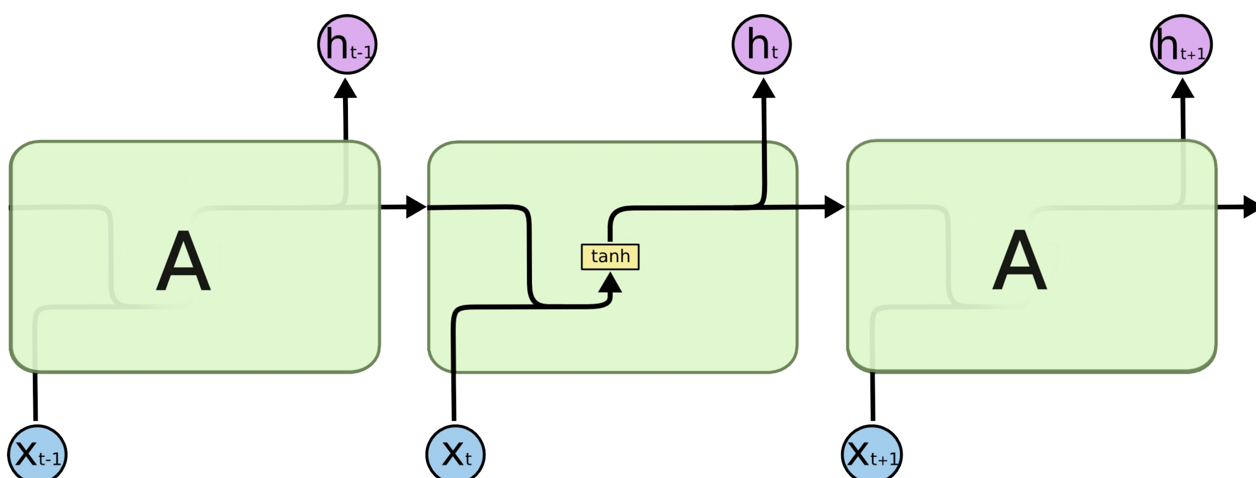
万幸的是，LSTM没有这个问题！

LSTM 网络

长短期记忆网络 - 通常简称为“LSTMs”，是一种特殊的RNN，适用于学习长期依赖。 他们由 Hochreiter 和 Schmidhuber（1997）介绍引入， 由许多其他的人们在后续的工作中重新定义和丰富。 他们在各种各样的问题中都工作的特别好，并且现在已经被广泛使用。

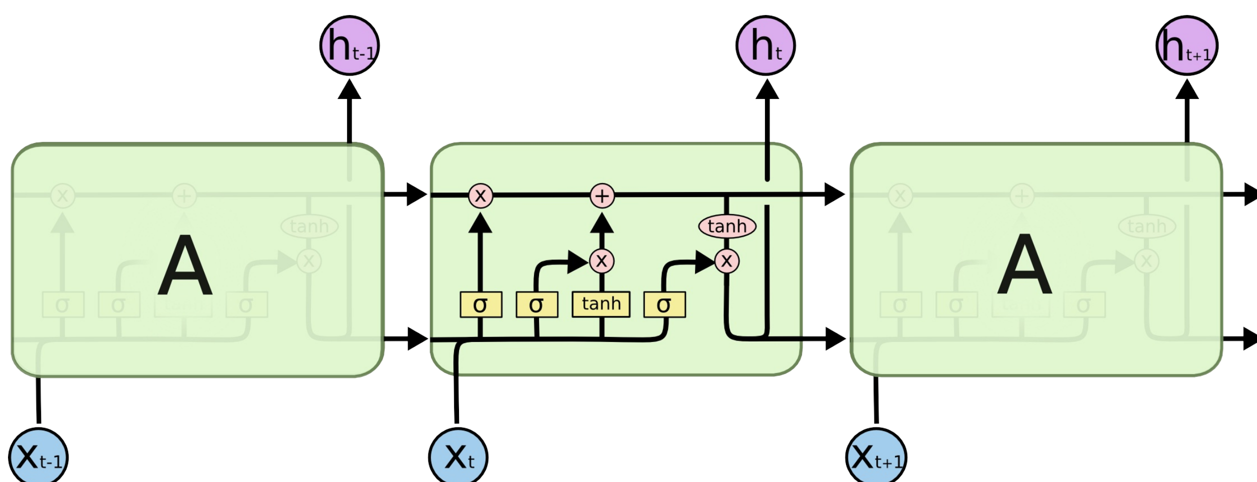
LSTMs 是为了避免长期依赖问题而特殊设计的。为长期时间记忆信息实际上是他们默认的行为， 而非他们需要学习的东西！

所有RNN都有重复神经网络模型的链式形式。在标准的RNN中，这种重复模型会有一种非常简单的结构，比如简单的tanh层。



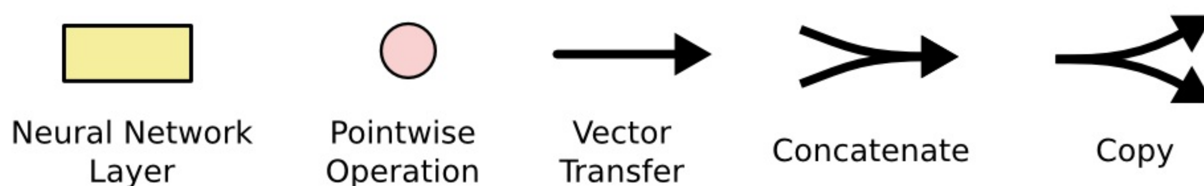
The repeating module in a standard RNN contains a single layer.

LSTM也有这种链式结构，但重复单元有着一种不同的结构。里面不再是只有单一的神经网络层，里面有四个，以非常简单的方式起作用。



The repeating module in an LSTM contains four interacting layers.

不要担心内部的细节。我们稍后会一步一步遍历LSTM图。现在，我们要熟悉我们将要使用的定义：

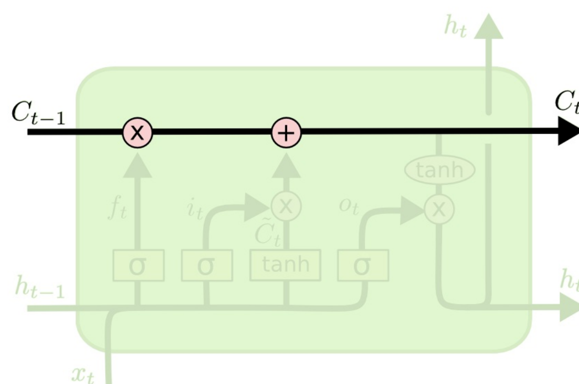


在上面的图中，每行都有一个箭头，从一个结点的输出到另外的结点的输入。粉色的圆代表结点操作，比如向量相加，而黄色的长方形是学习的神经网络层。 线的合并代表denote的链接，而箭头的分叉代表内容复制后流向不同的位置。

LSTM背后的核心思想

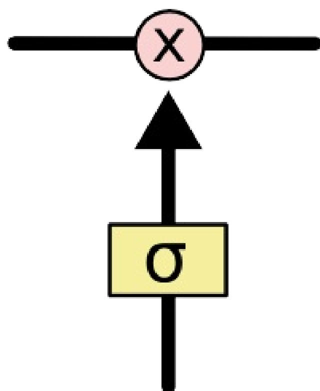
LSTM的关键在于cell的状态，也就是图中贯穿顶部的那条水平线。

cell的状态像是一条传送带，它贯穿整条链，其中只发生一些小的线性作用。信息流过这条线而不改变是非常容易的。



LSTM确实有能力移除或增加信息到cell状态中，由被称为门的结构精细控制。

门是一种让信息可选地通过的方法。它们由一个sigmoid神经网络层和一个点乘操作组成。



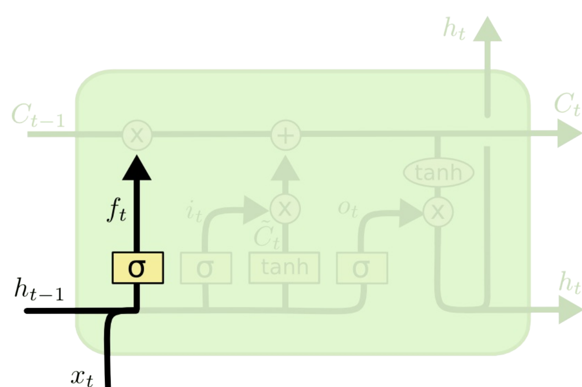
sigmod层输出[0, 1]区间内的数，描述了每个部分中应该通过的比例。输出0意味着“什么都不能通过”，而输出1意味着“让所有东西通过！”。

一个LSTM有四个这样的门，以保护和控制cell的状态。

深入浅出LSTM

我们的LSTM的第一步是决定我们需要从cell状态中扔掉什么样的信息。这个决策由一个称为“遗忘门”的sigmoid层做出。它观察h_{t-1}和x_t，位cell状态C_{t-1}中每个number输出一个0和1之间的数。1代表“完全保留这个值”，而0代表“完全扔掉这个值”。

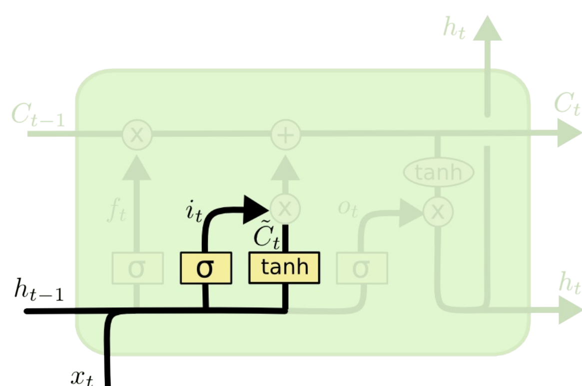
让我们回到我们那个基于上文预测最后一个词的语言模型。在这样一个问题中，cell的状态可能包含当前主题的种类，这样才能使用正确的名词。当我们看到一个新的主题的时候，我们会想要遗忘旧的主题的种类。



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

下一步是决定我们需要在cell state里存储什么样的信息。这个问题有两个部分。第一，一个sigmoid层调用“输入门”以决定哪些数据是需要更新的。然后，一个tanh层为新的候选值创建一个向量C~t，这些值能够加入state中。下一步，我们要将这两个部分合并以创建对state的更新。

在我们的语言模型的例子中，我们想要把主题的种类加入到cell state中，以替代我们要遗忘的旧的种类。

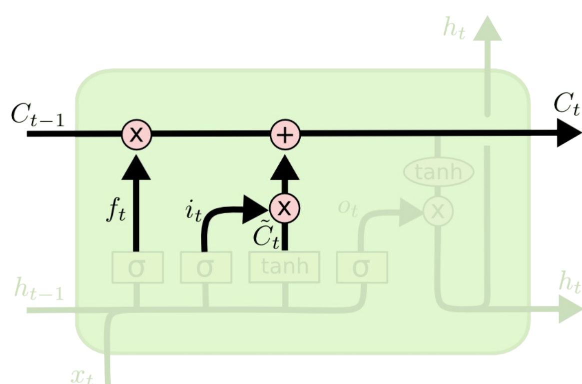


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

现在是时候更新旧的cell stateC_{t-1}到新的cell stateC_t。前一步已经决定了我们需要做的事情，我们只需要实现它。

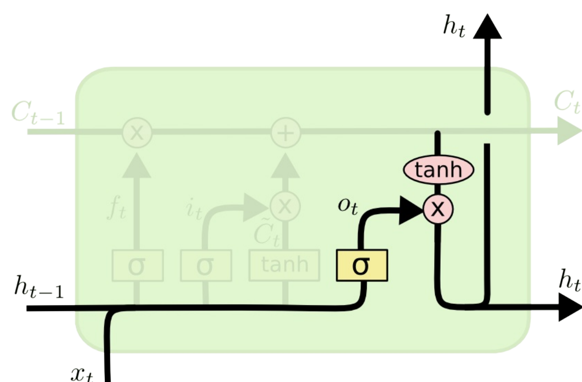
我们把旧的state与f_t相乘，遗忘我们先前决定遗忘的东西，然后我们加上i_t * C~t。这是新的候选值，受我们对每个状态值的更新度约束而缩放。

在语言模型的例子中，这就是我们真正扔掉旧主题种类，并增加新的信息的地方，正如我们之前所决定的。



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

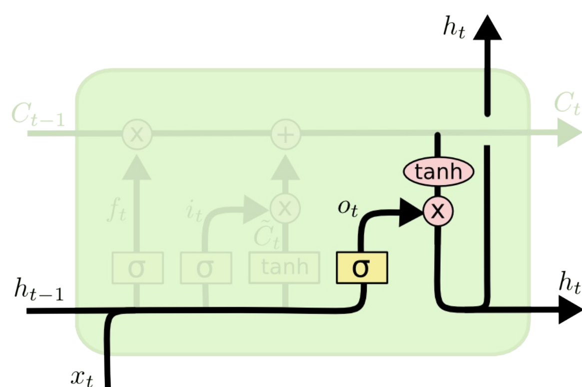
最后，我们需要决定要输出的东西。这个输出基于我们的cell state，但会是一个过滤版本。首先，我们运行一个sigmoid层，以决定cell state中的那个部分是我们将要输出的。然后我们把cell state放进tanh（将数值压到-1和1之间），最后将它与sigmoid门的输出相乘，这样我们就只输出了我们想要的部分了。



$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

语言模型的例子中，由于它仅关注一个主题，它可能会输出与一个动词相关的信息，以防后面还有其他的词。比如，它可能输出这个主题是单数还是复数，让我们知道如果后面还有东西，动词才会对应出现。



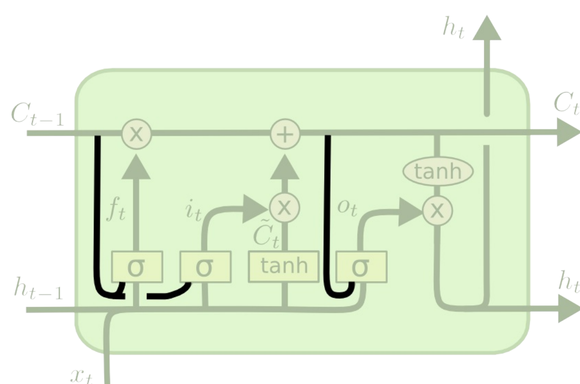
$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM变种

到目前为止我所描述的是一种非常普通的LSTM，但不是所有的LSTM都和上面描述的这种一样。事实上，几乎所有涉及LSTM的文章用的版本都稍有不同，差别微小，但值得一谈。

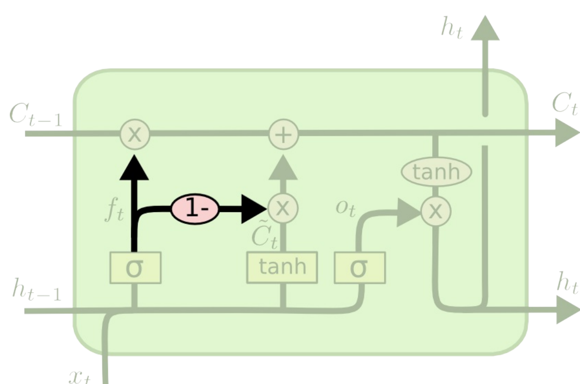
一种由Gers & Schmidhuber (2000)介绍的广受欢迎的LSTM变种，添加了“门镜连接”。这意味着我们可以让门观察cell状态。



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \ + \ b_o\right)$$

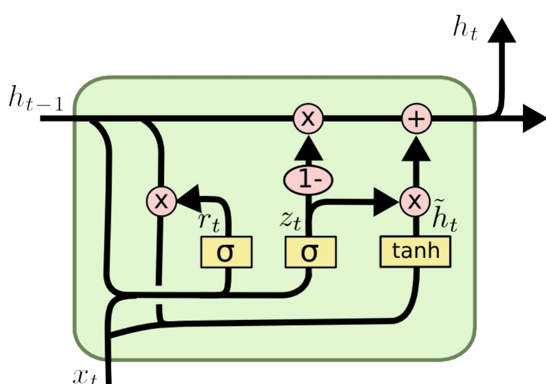
上面的图为每个门都添加了门镜，但许多文章只会给一部分门镜。

另一种变种是使用多个遗忘门和输入门。我们不再分别判断该遗忘和添加的东西，我们同时做出决策。我们只在填充某个位置的时候遗忘原来的东西，我们值在遗忘某些东西的时候输入新的数据。



$$C_t = f_t * C_{t-1} + \boldsymbol{(1 - f_t)} * \tilde{C}_t$$

一个稍微更奇特的变种是循环门单元（Gated Recurrent Unit，GRU），由 Cho, et al. (2014)提出。它组合了遗忘门和输入门到一个单独的“更新门”中。它也合并了cell state和hidden state，并且做了一些其他的改变。结果模型比标准LSTM模型更简单，并且正越来越受欢迎。



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

A gated recurrent unit neural network.

这些只是一些最值得一提的LSTM变种。还有许多其他种类，像Yao, et al. (2015)提出的Depth Gate RNN。也有许多复杂的不同方法来处理长期依赖，像 Koutnik, et al. (2014)提出的Clockwork RNN。

哪种变种是最好的？这些区别重要吗？ Greff, et al. (2015)对流行的变种做了一个很好的比较，发现它们都是一样的。Jozefowicz, et al. (2015)测试了超过一万中RNN结构，发现某些任务情形下，有些比LSTM工作得更好。

结论

首先，我讲述了人们用RNN获得的巨大成果。而这些成果都用到了LSTM，它们在大多数任务中都工作得好得多！

列出方程的话，LSTM看起来很吓人。幸好，在这篇文章里一步步看下来让它们变得相对可以接受了些。

LSTM是我们在RNN上取得的一大步。我们自然会想：还有另一个突破口吗？研究人员中的一个通常的观点是“有！是注意力！”思路是让一个RNN收集信息的每一步都关注更大的一个信息。例如，如果你用一个RNN抽取图片信息来描述它，RNN可能可以为每个输出的词都从图片拿一部分进行分析。事实上，Xu, et al. (2015)就是这样做的 - 这可能是一个有趣的出发点，如果你想要探索注意力这个话题的话。已经有许多令人惊艳的成果了，并且似乎还有更多不为人知的研究。

注意力不是RNN研究中唯一刺激的线。例如，网格LSTM（Kalchbrenner, et al. (2015)）,生产模型中使用RNN（ Gregor, et al. (2015), Chung, et al. (2015), or Bayer & Osendorfer (2015)），都很有趣。最近几年是RNN的黄金时代，下一年更是如此。

致谢

略

TensorFlow学习笔记

- TensorFlow安装
- 初识Tensorboard
- SKflow

> 觉得我的文章对您有帮助的话，就给个star吧～

土豪可以打赏支持，一分也是爱



TensorFlow 安装踩坑日志

Install TensorFlow

安装教程就在TensorFlow的官网上>>>点击查看

按照官方的流程装就好了，这里讲一下几种方式的特点：

1. pip: 安装在全局的python解释器中，简单
2. Third party: Virtualenv, Anaconda and Docker：都能创建tensorflow独立的编译环境，但就是多了一份包
3. Source: 能够适应不同的python版本（比如编译一个3.5版的），但源码编译可能有许多坑
4. ubuntu安装时，需要注意自己的python - pip - tensorflow版本是否对应（比如是否都是2.7），
5. 使用sudo命令时，注意自己的环境变量是否变化（会导致pip或python命令对应的版本变化）
6. 具体讲一下ubuntu安装tensorflow流程：
   - 安装anaconda2
   - 确定自己终端的pip和python版本：

     ```
     $ pip -V && python -V
     ```

     确认使用的是否都来自anaconda，如果不是，则应该使用类似这样的命令运行对应的pip：

     ```
     $ /home/cwh/.conda/envs/tensorflow/bin/pip -V
     ```

   即最好安装到tensorflow自己的python环境里，不跟anaconda原来的环境混淆

   使用sudo命令时最好也看一下版本

   - 使用anaconda创建一个tensorflow虚拟环境：

     ```
     $ conda create -n tensorflow python=2.7
     ```

   - 切换到tensorflow环境下（实际上是更换了环境变量里的pip和python），下载安装tensorflow，需要sudo权限

```
$ source activate tensorflow
(tensorflow)$ sudo pip install --ignore-installed --upgrade https://storage.g
oogleapis.com/tensorflow/linux/cpu/tensorflow-0.8.0rc0-cp27-none-linux_x86_64
.whl
$ source deactivate
```

注意如果安装的是gpu版本，还需要按照官网说明安装cuda和cudaCNN，具体教程看这个视频，不能科学上网的访问这个地址，注意一下你的显卡算力

- 如果pip安装速度慢，不要换pip源，复制whl名字，去谷歌一搜，找到对应的whl下下来，然后pip install xxx.whl，整个过程比全pip安装要快得多
- 如果setuptools安装失败，报"Cannot remove entries from nonexistent file"，就要用

  ```
  $ pip install --ignore-install setuptools
  ```

  覆盖安装
- 安装成功后就可以在tensorflow的python环境下，执行import tensorflow看看了。

初识Tensorboard

官方教程：https://www.tensorflow.org/versions/master/how_tos/graph_viz/index.html

> TensorFlow自带的一个强大的可视化工具

功能

这是TensorFlow在MNIST实验数据上得到Tensorboard结果

- Event: 展示训练过程中的统计数据（最值，均值等）变化情况
- Image: 展示训练过程中记录的图像
- Audio: 展示训练过程中记录的音频
- Histogram: 展示训练过程中记录的数据的分布图

原理

- 在运行过程中，记录结构化的数据
- 运行一个本地服务器，监听6006端口
- 请求时，分析记录的数据，绘制

实现

在构建graph的过程中，记录你想要追踪的Tensor

```
with tf.name_scope('output_act'):
    hidden = tf.nn.relu6(tf.matmul(reshape, output_weights[0]) + output_biases)
    tf.summary.histogram('output_act', hidden)
```

其中，

- summary.histogram用于生成分布图，也可以用scalar_summary记录存数值
- 使用scalar_summary的时候，tag和tensor的shape要一致
- name_scope可以不写，但是当你需要在Graph中体现tensor之间的包含关系时，就要写了，像下面这样：

```
with tf.name_scope('input_cnn_filter'):
    with tf.name_scope('input_weight'):
        input_weights = tf.Variable(tf.truncated_normal(
            [patch_size, patch_size, num_channels, depth], stddev=0.1), name='input_we
ight')
        variable_summary(input_weights)
    with tf.name_scope('input_biases'):
        input_biases = tf.Variable(tf.zeros([depth]), name='input_biases')
        variable_summary(input_weights)
```

- 在Graph中会体现为一个input_cnn_filter，可以点开，里面有weight和biases
- 用summary系列函数记录后，Tensorboard会根据graph中的依赖关系在Graph标签中展示对应的图结构
- 官网封装了一个函数，可以调用来记录很多跟某个Tensor相关的数据：

```
def variable_summary(var):
    """Attach a lot of summaries to a Tensor (for TensorBoard visualization)."""
    with tf.name_scope('summaries'):
        mean = tf.reduce_mean(var)
        tf.summary.scalar('mean', mean)
        with tf.name_scope('stddev'):
            stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
        tf.summary.scalar('stddev', stddev)
        tf.summary.scalar('max', tf.reduce_max(var))
        tf.summary.scalar('min', tf.reduce_min(var))
        tf.summary.histogram('histogram', var)
```

- 只有这样记录国max和min的Tensor才会出现在Event里面
- Graph的最后要写一句这个，给session回调

```
merged = tf.summary.merge_all()
```

Session 中调用

- 构造两个writer，分别在train和valid的时候写数据：

```
train_writer = tf.summary.FileWriter(summary_dir + '/train',
                                     session.graph)
valid_writer = tf.summary.FileWriter(summary_dir + '/valid')
```

- 这里的summary_dir存放了运行过程中记录的数据，等下启动服务器要用到
- 构造run_option和run_meta，在每个step运行session时进行设置：

```
summary, _, l, predictions =
    session.run([merged, optimizer, loss, train_prediction], options=run_options, feed
_dict=feed_dict)
```

- 注意要把merged拿回来，并且设置options
- 在每次训练时，记一次：

```
train_writer.add_summary(summary, step)
```

- 在每次验证时，记一次：

```
valid_writer.add_summary(summary, step)
```

- 达到一定训练次数后，记一次meta做一下标记

```
train_writer.add_run_metadata(run_metadata, 'step%03d' % step)
```

查看可视化结果

- 启动TensorBoard服务器：

```
python安装路径/python TensorFlow安装路径/tensorflow/tensorboard/tensorboard.py --logdir=p
ath/to/log-directory
```

注意这个python必须是安装了TensorFlow的python，tensorboard.py必须制定路径才能被python找到，logdir必须是前面创建两个writer时使用的路径

比如我的是：

```
/home/cwh/anaconda2/envs/tensorflow/bin/python /home/cwh/anaconda2/envs/tensorflow/lib
/python2.7/site-packages/tensorflow/tensorboard/tensorboard.py --logdir=~/coding/pytho
n/GDLnotes/src/convnet/summary
```

如果默认python解释器就是包含tensorflow的python解释器，可以直接输入

```
tensorboard --logdir=path/to/log/dir
```

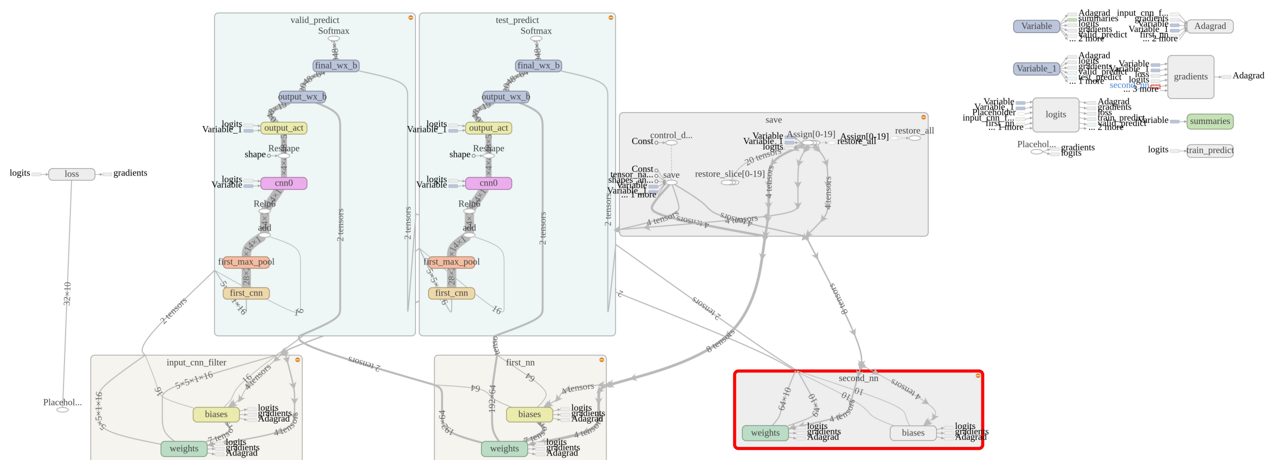
使用python

- 然后在浏览器输入 http://127.0.0.1:6006 就可以访问到tensorboard的结果

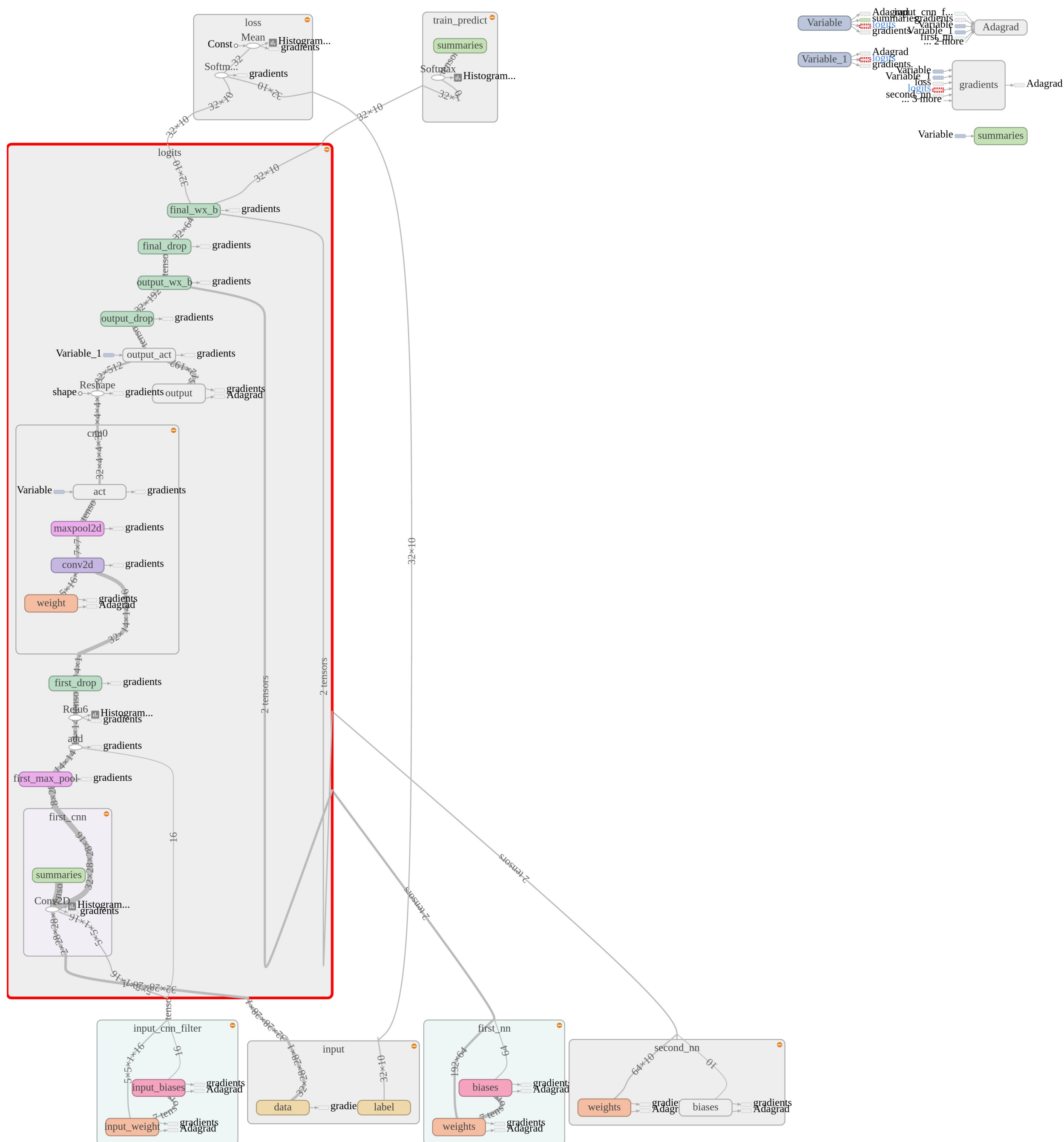
强迫症踩坑后记

- 之前我的cnn代码里有valid_prediction，所以画出来的graph有两条分支，不太清晰，所以只留了train一个分支

修改前：


valid_predict
test_predict
save
input_cnn_filter
first_nn
second_nn
loss
logits
gradients
Softmax
final_wx_b
output_wx_b
output_act
Reshape
cnn0
Relu
add
first_max_pool
first_cnn
biases
weights
Variable
Variable_1
Adagrad
summaries
train_predict
restore_all
Placehol...

修改后：



- 多用with，进行包裹，这样才好看，正如官网说的，你的summary代码决定了你的图结构
- 不是所有的tensor都有必要记录，但是Variable和placeholder最好都用summary记录一下，也是为了好看
- 由于有了gradient的计算，所以与gradient计算相关的都会被拎出来，下次试一下用其他optimizer

我的CNN TensorBoard代码：cnn_board.py

参考资料

- mnist_with_summaries.p

skflow

sklearn风格的api，用tensorflow来处理训练sklearn的数据集，现在已经合并到tf.contrib里。

```
# tensorflow 0.8
import skflow
from sklearn import datasets, metrics

iris = datasets.load_iris()
classifier = skflow.TensorFlowDNNClassifier(hidden_units=[10, 20, 10], n_classes=3)
classifier.fit(iris.data, iris.target)
score = metrics.accuracy_score(iris.target, classifier.predict(iris.data))
print("Accuracy: %f" % score)
```

分布式TensorFlow

> code : distrib.py

- 然后在不同的机器上启动之：

```
# On ps0.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=ps --task_index=0
# On ps1.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=ps --task_index=1
# On worker0.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=worker --task_index=0
# On worker1.example.com:
$ python trainer.py \
     --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
     --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
     --job_name=worker --task_index=1
```